Learning Hard C# 博客原文

wizardforcel

Published with GitBook



目錄

Learning Hard C# 博客原文

作者：天平Learning

来源：Learning hard

作者的新书《Learning hard C#学习笔记》已出版，开发者们可以购买此书来支持作者。

Learning Hard C# 中的一系列文章涵盖了C#的核心特性、开发技巧、设计模式、并发与网络编程、WPF、WCF、Asp.NET入门等一系列进阶话题，有助于开发者们深入理解C#，可以和现有的编程书相互补充。

C# 基础知识系列

C#基础知识系列终于告了一个段落了, 本系列中主要和大家介绍了C#1.0到C# 4.0中一些重要的特性，刚开始写这个专题的初衷主要是我觉得掌握了C#这些基础知识之后，对于其他任何的一门语言都是差不多的，这样可以提高朋友们对其他语言的掌握，以及可以让大家更加迅速地掌握.NET的新特性， 并且相信这个系列对于找工作的朋友也是很有帮助的，因为很多公司面试都很看重基础知识是否扎实，是否对C#有一个全面的认识和理解，所以很多公司面试都会问到一些C#基础概念的问题，例如，经常面试会问：你是如何理解委托的，如何理解匿名函数等问题。

然而这个系列中并没有介绍COM互操作性的内容以及.Net 4.5中的一些新特性，所以后面将会对这两个方面的内容进行补充，由于这个系列托的太久了(大概也有3个月吧)，所以就先告一段落的，后面将会带来.NET互操作性系列的介绍。下面就为这个系列文章做一个索引，方便大家收藏和查找。

C#基础知识系列索引

C#1.0

1. 深入解析委托——C#中为什么要引入委托
2. 委托本质论
3. 如何用委托包装多个方法——委托链
4. 事件揭秘
5. 当点击按钮时触发Click事件背后发生的事情

C# 2.0

6. 泛型基础篇——为什么引入泛型
7. 泛型深入理解(一)
8. 泛型深入理解(二)
9. 深入理解泛型可变性
10. 全面解析可空类型
11. 匿名方法解析
12. 迭代器

C# 3.0

13. 全面解析对象集合初始化器、匿名类型和隐式类型
14. 深入理解Lambda表达式
15. 全面解析扩展方法

16. Linq介绍

C# 4.0

17. 深入理解动态类型

C# 5.0

18. C# 5.0 新特性——Async和Await使异步编程更简单

从C#的所有特性可以看出,C#中提出的每个新特性都是建立在原来特性的基础上,并且是对原来特性的一个改进, 做这么多的改进主要是为了方便开发人员更好地使用C#来编写程序,是让我们写更少的代码来实现我们的程序,把一些额外的工作交给编译器去帮我们做,也就是很多人说微软很喜欢搞语法糖的意思(语法糖即让编译器帮我们做一些额外的事情，减少开发人员所考虑的事情，使开发人员放更多的精力放在系统的业务逻辑上面。)，大家从C# 3中提出的特性中可以很好的看出这点(指的是玩语法糖)，C#3中几乎大部分特性都是C#提供的语法糖，从CLR层面来说(指的是增加新的IL指令)，C# 3并没有更新什么，C# 4中提出的动态类型又是建立在表达式树的基础上，包括Linq也是建立在表达式树的基础上，所以每个特性都是层层递进的一个关系。相信C#后面提出的新特性将会更加方便我们开发程序，感觉所有语言的一个统一的思想都是——写更少的代码，却可以做更多的事情。但是我们不能仅仅停住于知道怎么使用它，我们还应该深入研究它的背后的故事，知道新特性是如何实现的和原理。用一句说就是——我们要知其然之气所以然，学习知识应该抱着刨根问底的态度去学习,相信这样的学习方式也可以让大家不感到心虚,写出的程序将会更加自信。

[C# 基础知识系列]专题一：深入解析委托——C#中为什么要引入委托

引言：

对于一些刚接触C# 不久的朋友可能会对C#中一些基本特性理解的不是很深，然而这些知识也是面试时面试官经常会问到的问题，所以我觉得有必要和一些接触C#不久的朋友分享下关于C#基础知识的文章，所以有了这个系列，希望通过这个系列让朋友对C#的基础知识理解能够更进一步。然而委托又是C#基础知识中比较重要的一点，基本上后面的特性都和委托有点关系，所以这里就和大家先说说委托，为什么我们需要委托。

一、**C#**委托是什么的？

在正式介绍委托之前，我想下看看生活中委托的例子——生活中，如果如果我们需要打官司，在法庭上是由律师为我们辩护的，然而律师真真执行的是当事人的陈词，这时候律师就是一个委托对象，当事人委托律师这个对象去帮自己辩护。这就是我们生活中委托的例子的。然而C#中委托的概念也就好比律师对象（从中可以得出委托是一个类，，因为只有类才有对象的概念，从而也体现了**C#**是面向对象的语言）。

介绍完生活中委托是个什么后，现在就看看C#中的委托怎样和生活中的对象联系起来的，C#中的委托相当于C++中的函数指针（如果之前学过C++就知道函数指针是个什么概念的了），函数指针是用指针获取一个函数的入口地址，然后通过这个指针来实现对函数的操作。C#中的委托相当于C++中的函数指针，也就说两者是有区别的：委托是面向对象的，类型安全的，是引用类型（开始就说了委托是个类），所以在使用委托时首先要 定义——>声明——>实例化——>作为参数传递给方法——>使用委托。下面就具体看下如何使用委托的：

一、定义：delegate void Mydelegate(type1 para1,type2 para2);

二、声明： Mydelegate d;

三、实例化：d =new Mydelegate(obj.InstanceMethod);(把一个方法传递给委托的构造器)，前面三步就好比构造一个律师对象，方法InstanceMethod好比是当事人

四、作为参数传递给方法：MyMethod（d）;（委托实现把方法作为参数传入到另一个方法，委托就是一个包装方法的对象）

五、在方法中使用委托。MyMethod方法好比是法官，MyMethod方法先调用委托，委托在调用方法InstanceMethod,这个过程就如法官向律师问话，然后律师之前肯定向当事人了解了案件的情况。C#委托中好比是律师，真真诉说案情的是当事人（真真被调用的是实例方法InstanceMethod）

MyMethod方法的定义如下：

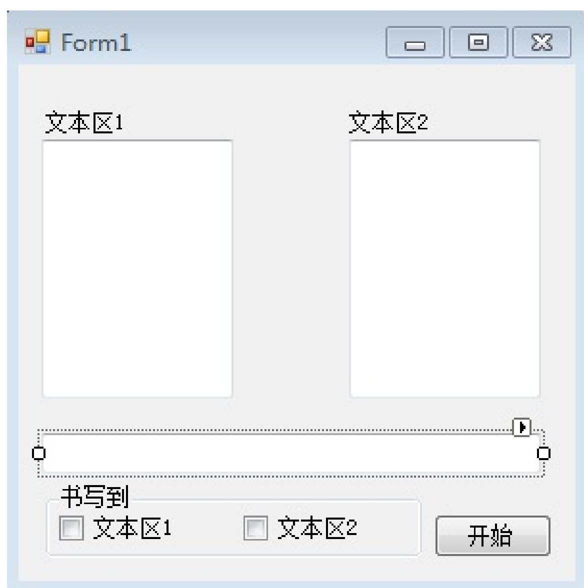
```
private void MyMethod(Mydelegate mydelegate)

{
    // 使用委托
    mydelegat(arg1,arg2);

}
```

二、**C#**中为什么要使用委托的？相信经过上面的介绍，大家应该对委托不再陌生了吧，然而我们为什么需要委托的，好好地为什么要实例化中间这个对象的，为什么不直接在MyMethod方法里面调用InstanceMethod方法的，这样不是自找麻烦的吗？为了大家可以更好的明白为什么要使用委托，下面通过一个Window Form的"文字抄写员"程序要解释下为什么。

程序实现的功能是：在下方文本框输入文字，勾选"书写到"组合框中的"文本区1"或"文本区2"复选框后点击"开始"按钮，程序会自动将文本框中的文字"抄写"到对应的文本区中去。程序界面如下：



传统的实现代码为：

```
namespace 文字抄写员
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            if (checkBox1.Checked == true)
            {
```

```
                textBox1.Clear();
                textBox1.Refresh();
                // 调用方法WriteRichTextBox1向文本区1写入文字
                this.WriteTextBox1();
                textBox3.Focus();
                textBox3.SelectAll();
            }
            if (checkBox2.Checked == true)
            {
                textBox2.Clear();
                textBox2.Refresh();
                // 调用方法WriteRichTextBox2向文本区2写入文字
                this.WriteTextBox2();
                textBox3.Focus();
                textBox3.SelectAll();
            }
        }

        private void WriteTextBox1()
        {
            string data = textBox3.Text;
            for (int i = 0; i < data.Length; i++)
            {
                textBox1.AppendText(data[i].ToString());
                //间歇延时
                DateTime now = DateTime.Now;
                while(now.AddSeconds(1)>DateTime.Now)
                { }
            }
        }

        private void WriteTextBox2()
        {
            string data = textBox3.Text;
            for (int i = 0; i < data.Length; i++)
            {
                textBox2.AppendText(data[i].ToString());
                //间歇延时
                DateTime now = DateTime.Now;
                while (now.AddSeconds(1) > DateTime.Now)
                {  }
            }
        }
    }
}
```

然而我们从代码中会发现WriteTextBox1()方法和WriteTextBox2()只有一行代码不一样的（ textBox1.AppendText(data[i].ToString()); 和 textBox2.AppendText(data[i].ToString());），其他都完全一样，而这条语句的差别就在于向其中写入文本的控件对象不一样，一个是TextBox1和TextBox2,现在这样代码是实现了功能，带式我们试想下，如果要实现一个写入的文本框不止2个，而

是好几十个甚至更多，那么不久要写出同样多数量的用于写入文本区的方法了吗？这样就不得不写重复的代码，导致代码的可读性就差，这样写代码也就是面向过程的一个编程方式，因为函数是对操作过程的一个封装，要解决这个问题，自然我们就想到面向对象 编程，此时我们就会想到把变化的部分封装起来，然后再把封装的对象作为一个对象传递给方法的参数的（这个思想也是一种设计模式——策略模式，关于设计模式系列会在后面也会给出的），下面就利用委托来重新实现下这个程序：

```
namespace 文字抄写员
{
    public partial class Form1 : Form
    {
        // 定义委托
        private delegate void WriteTextBox(char ch);
        // 声明委托
        private WriteTextBox writeTextBox;

        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            if (checkBox1.Checked == true)
            {
                textBox1.Clear();
                textBox1.Refresh();
                // 实例化委托
                writeTextBox = new WriteTextBox(WriteTextBox1);
                // 作为参数
                WriteText(writeTextBox);

                textBox3.Focus();
                textBox3.SelectAll();
            }
            if (checkBox2.Checked == true)
            {
                textBox2.Clear();
                textBox2.Refresh();
                // 实例化委托
                writeTextBox = new WriteTextBox(WriteTextBox2);
                // 作为参数
                WriteText(writeTextBox);

                textBox3.Focus();
                textBox3.SelectAll();
            }
        }

        private void WriteText(WriteTextBox writetextbox)
        {
```

```
            string data = textBox3.Text;
            for (int i = 0; i < data.Length; i++)
            {
                // 使用委托
                writetextbox(data[i]);
                DateTime now = DateTime.Now;
                while (now.AddSeconds(1) > DateTime.Now)
                { }
            }
        }

        private void WriteTextBox1(char ch)
        {
            textBox1.AppendText(ch.ToString());
        }
        private void WriteTextBox2(char ch)
        {
            textBox2.AppendText(ch.ToString());
        }
    }
}
```

引入委托后实现的代码中，我们通过WriteText方法来向文本区写入内容，它所执行的只是抽象的”写文本“操作，至于究竟像那个文本框写入文字，对于编写WriteText方法的程序来说是不知道，委托writeTextBox就像一个接口一样（面向对象设计原则中有一个很重要的原则就是——针对接口编程，不针对实现编程），屏蔽了操作对象的差别（方法到底是想向文本区1写入文本还是像文本区2写入文本，现在我方法里面不需要去关心，我只需要集中在实现”书写文本”这个操作，而不必纠结操作对象的选择）。

三、委托的作用到底是什么？——委托总结陈词

相信通过上面两部分大家也明白了委托是个什么东西以及C#中为什么要引入委托这个概念。现在就总结下引入委托后到底作用在那里的？从上面的委托代码中可以发现，引入委托后，编程人员可以把方法的引用封装在委托对象中（把过程的调用转化为对象的调用，充分体现了委托加强了面向对象编程的思想。），然后把委托对象传递给需要引用方法的代码，这样在编译的过程中我们并不知道调用了哪个方法，这样一来，C#引入委托机制后，使得方法声明和方法实现的分离，充分体现了面向对象的编程思想。

委托对自己的总结：

我是一个特殊的类，我定义了方法的类型，（就像int定义了数字类型一样，当用一个方法实例化委托对象时，这个委托就代表一个方法，这个方法的类型就是委托类型），我可以将方法当做另一个方法的参数来进行传递，使得程序更容易扩展

四、小结

写到这里本专题介绍的内容也结束了，在本专题中有些地方提到了一些设计模式的知识的，如果有朋友对设计模式还没有开始学习的话，建议大家都去学习下的，并且我也会在后面的系列中向大家分享下我的理解的。对于本系列的下一专题将和大

家分享下我理解的事件到底是个什么样的概念。最后希望本专题可以让大家进一步理解委托。

[C# 基础知识系列]专题二：委托的本质论

引言：

上一个专题已经和大家分享了我理解的——C#中为什么需要委托，专题中简单介绍了下委托是什么以及委托简单的应用的，在这个专题中将对委托做进一步的介绍的，本专题主要对委本质和委托链进行讨论。

一、委托的本质

平时我们很容易使用委托——用C# **delegate**关键字定义委托，再用**new**操作符构造委托实例，然后通过调用委托实例来调用回调方法（就是用一个了委托对象的变量来代替方法名，这句话如果刚接触的人不好理解的话，这里给个例子：MyDelegate mydelegate =new Mydelegate(obj.mymethod),**MyDelegate** 是定义的一个委托，假设定义的是没有参数的，然后调用委托实例是这样的——mydelegate()，大家可以发现此时调用委托和调用方法的方式是一模一样的，如果没有看前面**mydelegate**是个委托类型，大家都会认为这是直接调用一个方法，而不是调用委托实例，通过这个例子大家应该很容易明白了这句话了吧——用一个委托对象的变量来代替方法名)，相信通过括号内的讲解后，相信大家又会对委托有进一步的理解的——委托就是方法的代替品，委托变量此时着方法名，大家可以简单理解委托是方法的一个 “外号”。

前面的都介绍了委托的一些使用和理解的，现在就让我我们来进一步看看编译器和CLR在背后对我们用**delegate** 关键字定义的委托类型做了些什么事情的，前一个专题中我和大家说过委托是一个类，这么是有根据的，因为我们在IDE中定义一个委托类型时，最终是通过编译器将定义的代码转化为中间语言**IL,**然后再执行中间语言中的代码来转化为本机代码的，所以在**Visual Studio**中编写的代码只是一个包装而已，真真程序执行的是中间语言中的代码的。现在就看看编译器把我们定义的委托类型转化为什么样的中间语言代码的。

当我们在类中像下面这样定义一个委托时：

编译器把我们定义的委托类型编译成一个下面这样的类：

```
Public class DelegateTest: System.MulticastDelegate
 {
         public DelegateTest(Object object, IntPtr method);

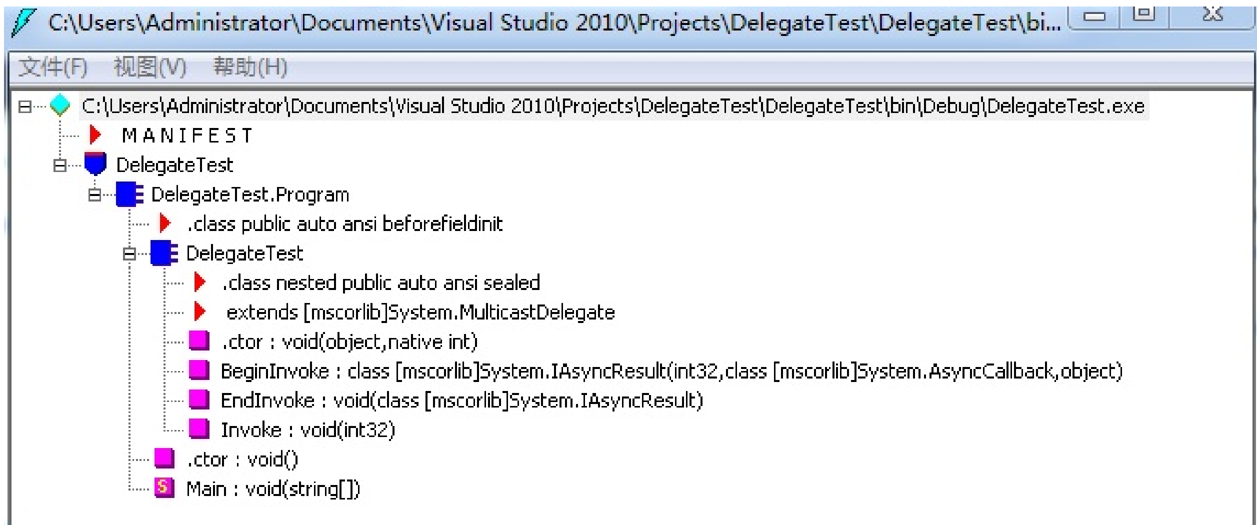
         public virtual Void Invoke(int32 parm);

         public virtual IAsyncResult BeginInvoke(Int32 parm, AsyncC

          public virtual void EndInvoke(IAsyncResult result);

 }
```

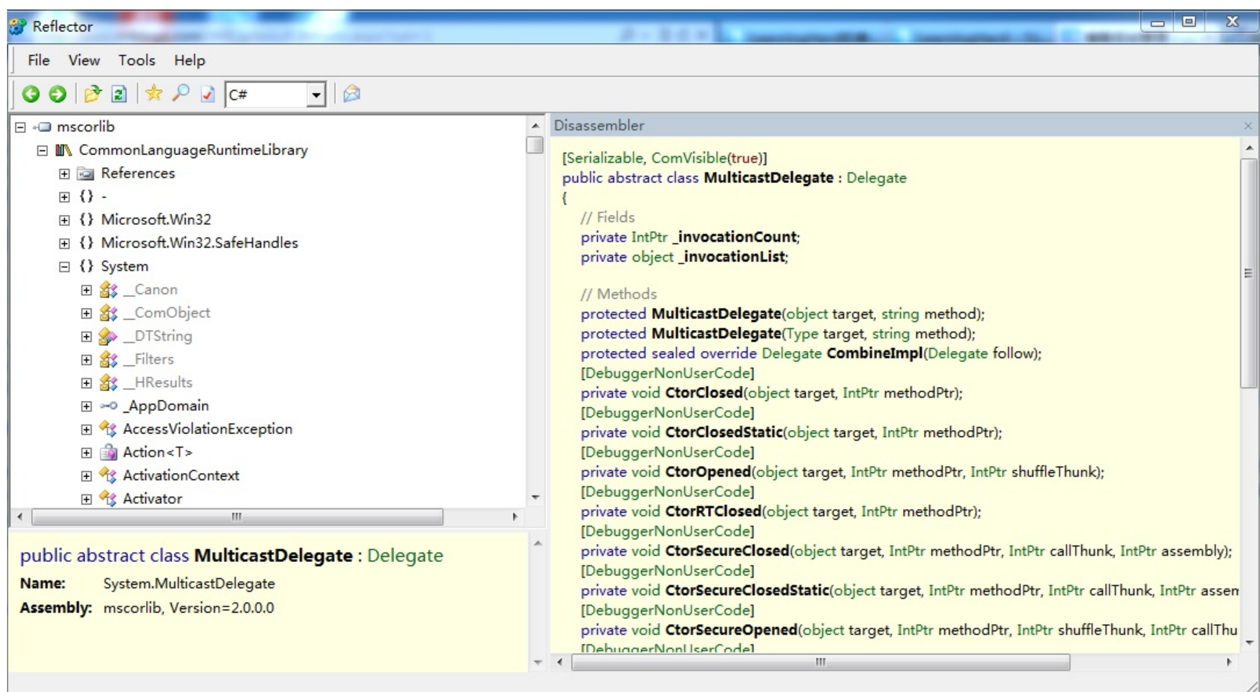
从中间语言的代码就可以很明显的看出我们在代码中写的委托，对于中间语言来说就是一个类，该类继承于FCL中定义的**Systme.MulticastDelegate**类型，所有委托类型都派生于**MulticastDelegate**，该类中还定义了四个方法，一个构造函数，**Invoke**方法，还有就是两个异步方法**BeginInvoke**和**EndInvoke**方法，关于这两个异步方法，大家可以查看我博客中的线程系列。大家可以用**ILDasm.exe**工具去查看委托生成的中间代码，下面我截的一个图（从我们定义的**DelegateTest**的前面的图标和我们主程序传递**Program**的图标是一样的，然而**Program**是一个类，很明显定义的委托**DelegateTes**也是一个类的）：



由于所有委托类型都是继承于**MulticastDelegate**,**MulticastDelegate**又继承于**Delegate**,所以委托类型继承了MulticastDelegate的字段、属性和方法，在这些成员中，有三个非公共字段与后面专题要介绍的委托链有关，所以在这里先列出来的：

字段	类型	解释
_target	**System.Object**	当委托对象包装的是一个静态方法时，这个字段为null,当委托对象包装一个实例方法时，这个字段引用的是方法所在的类的对象
_methodPtr	**System.IntPtr**	一个内部的整数，可以认为是方法句柄，标识着要调用的方法
_invocationList	**System.Object**	该字段通常为null，当构造一个委托链（多播委托）时，才引用一个委托数组。具体下一部分讲解。

大部分人可能会有这么个疑问，既然是非公共字段，所以在MSDN上是看不到的，那我是怎么知道有这三个字段的呢？大家可以通过**Reflector**工具是反编译查看源码，**Multicastdelegate** 类通过MSDN查找可以知道该类的命名空间和程序集，这样就可以更具程序集和命名空间用Reflector工具查看**Multicastdelegate**类的源码，下面是我用**Reflector**这个工具查看到的源码截图：



从截图中可以看出**MulticastDelegate** 类中只有两个字段，却没有前面表格中列出的***methodPtr***和**target**字段的，这两个字段是定义在**Delegate**类中，大家使用**Reflector**工具来查看的，这里就不具体贴图了，文章最后会给出**Reflector**工具下载链接的。

委托对象就是一个包装器，包装了一个方法和调用该方法时要操作的对象，例如，执行下面的代码时：

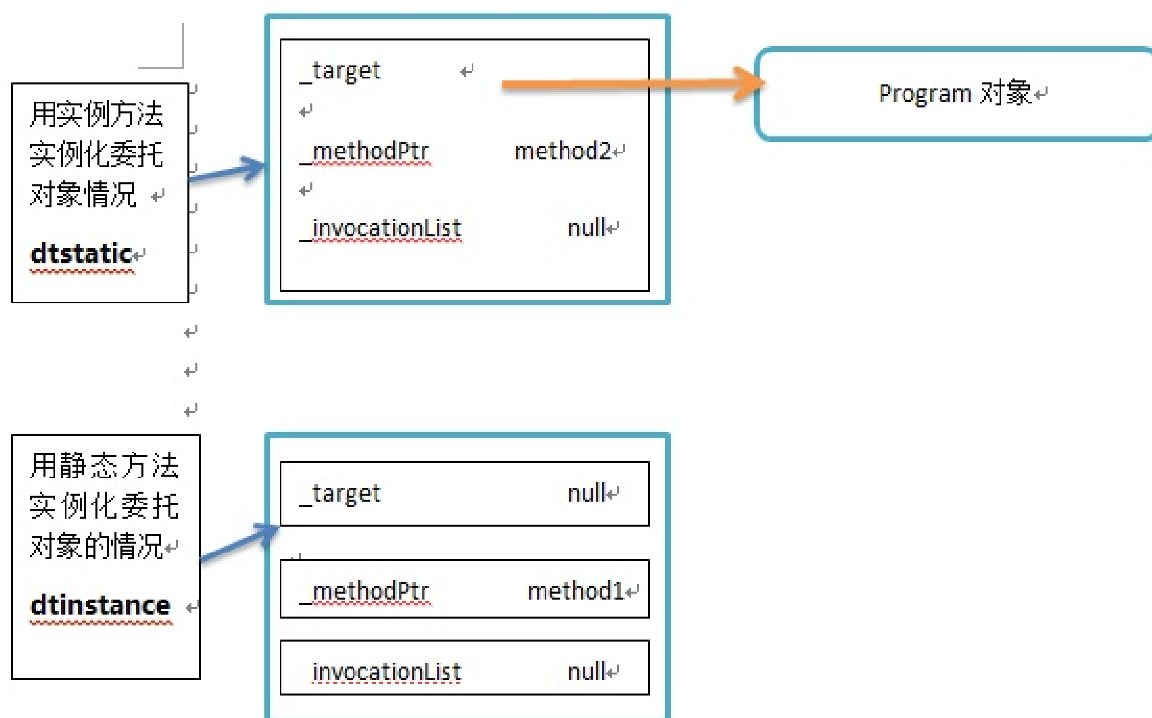
```
public  class Program
  {
     // 声明一个委托类型，它的实例引用一个方法
     // 该方法回去一个int 参数，返回void类型
     public delegate void DelegateTest(int parm);

      public static void Main(string[] args)
      {
          // 用静态方法来实例化委托
          DelegateTest dtstatic = new DelegateTest(Program.methoc

          // 用实例方法来实例化委托
          DelegateTest dtinstance = new DelegateTest(new Program(
      }
      private static void method1(int parm)
      {
          Console.WriteLine("调用的是静态方法，参数值为：" + parm);
      }

      private void method2(int parm)
      {
          Console.WriteLine("调用的是实例方法，参数值为：" + parm);
      }
  }
```

代码中dtstatic 和dtinstance变量引用了初始化好的**DelegateTest**委托对象，此时这两个委托对象的上面列出来的三个字段初始化情况如下图：



二、总结

本专题从中间语言的角度去详细解析定义的委托类型经编译器转化后的的中间语言是怎样来解释一个委托类型的，得到的结论是——委托实际上是一个类，该类派生于**MulticastDelegate**类，且继承了该类的_target,_methodPtr和_invocationList这三个字段，当我们初始化一个委托对象时，此时就会先初始化这三个字段，对于包装实例方法和静态方法的委托，初始化这三个字段也有所不一样，在上面的截图中也所体现，这里引用了一个很重要的字段——_invocationList(即委托实例的调用列表)，对于委托对象包装一个方法时，该字段为null，如果委托对象要包装多个方法时，此时_invocationList字段就会被初始化为引用一个委托对象的数组（就是指向委托对象的一个集合），具体这方面的内容将在下一专题介绍委托链中为大家详细介绍。 到这里，本专题的内容也结束了，希望通过本专题，大家可以更进一步的理解C#中的委托。

Reflector工具的下载地址：http://files.cnblogs.com/zhili/Reflector.zip ,看完后觉得有帮助的话，请大家多多推荐下的，谢谢大家的支持。

[C# 基础知识系列]专题三：如何用委托包装多个方法——委托链

引言：

上一专题介绍了下编译器是如何来翻译委托的，从中间语言的角度去看委托，希望可以帮助大家进一步的理解委托，然而之前的介绍都是委托只是封装一个方法，那委托能不能封装多个方法呢？因为生活中经常会听到，我代表大家的意见等这样的说话，既然委托也是一个代表，那他如果只能代表一个人，那他的魅力就不是很大了吧，所以我们就会委托能不能代表多个方法的？ 答案是可以的，这就是本专题要讲的内容——委托链，委托链也是一个委托，只是因为它是把多个委托链在一起，所以我们就以委托链来这么称呼它的。

一、到底什么是委托链

我们平常实例化委托对象时都是绑定一个方法的， 前一个专题介绍的委托也是包装了一个方法的， 用前面的例子就是委派律师的只有一个人，也就是当事人只有一个的，但是现实生活中显然不是这样的，在官司的时候律师可以同时接多个案子，也是接收多个当时人的委派，这样，该律师就与多个当事人绑定在一起了， 需要了解多个当事人的案件情况的。其实这就是生活中的委托链，此时这位律师不仅仅是一个人的代表律师了，而是多个当事人的律师。生活中的委托链和C#中的委托链很类似的，现在就说说C#中的委托链到底是个什么的？

首先委托链就是一个委托，所以大家不要看到委托链感觉又是什么C#中的新特性的，然而要把多个委托链在一起，就必须存储多个委托的引用，那委托链对象是在哪里存储多个委托的引用的呢？还记得我们上一专题中，我们介绍的委托类型有三个非公共字段的吗？这三个字段是——_target,methodPtr 和_invocationList,至于这三个字段具体代表什么大家可以查看我的上一专题的文章，然而**_invocationList**字段正是存储多个委托引用的地方的。

为了更好的解释_invocationList是如何来存储委托引用的，下面先看一个委托链的例子和运行结果，然后再分析原因：

```
using System;

namespace DelegateTest

{
    public class Program
    {
        // 声明一个委托类型，它的实例引用一个方法
        // 该方法回去一个int 参数，返回void类型
        public delegate void DelegateTest(int parm);

        public static void Main(string[] args)
        {
            // 用静态方法来实例化委托
            DelegateTest dtstatic = new DelegateTest(Program.methoc

            // 用实例方法来实例化委托
            DelegateTest dtinstance = new DelegateTest(new Program(

            // 隐式调用委托
            dtstatic(1);

            // 显式调用Invoke方法来调用委托
            dtinstance.Invoke(1);

            // 隐式调用委托
            dtstatic(2);

            // 显式调用Invoke方法来调用委托
            dtinstance.Invoke(2);
            Console.Read();
        }
        private static void method1(int parm)
        {
            Console.WriteLine("调用的是静态方法，参数值为：" + parm);
        }

        private void method2(int parm)
        {
            Console.WriteLine("调用的是实例方法，参数值为：" + parm);
        }
    }

}
```

运行结果：

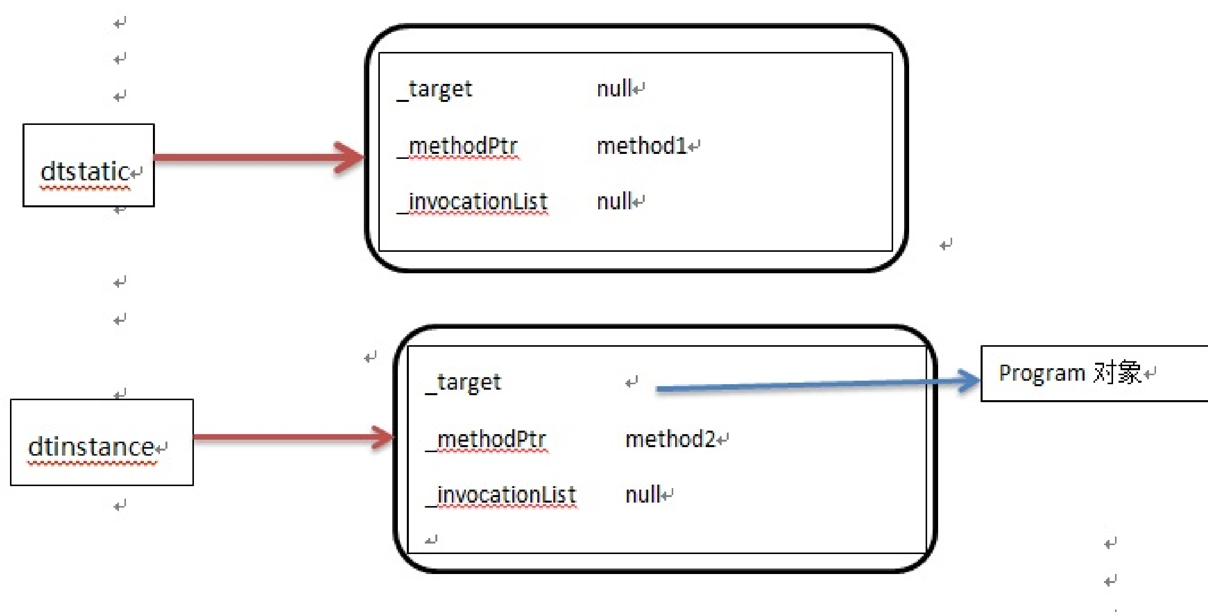
file:///c:/users/administrator/documents/visual studio 2010/Projects/DelegateChainDemo/...

```
调用的是静态方法, 参数值为: 1
调用的是实例方法, 参数值为: 1
调用的是静态方法, 参数值为: 2
调用的是实例方法, 参数值为: 2
```

下面就来分析下为什么会出现这样的结果的：

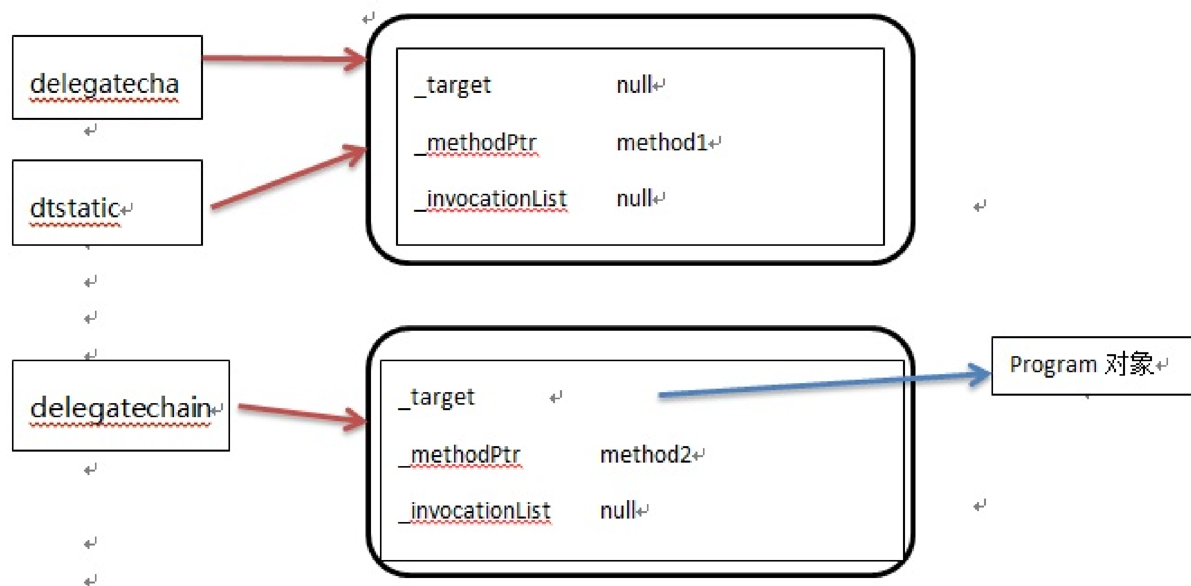
一开始我们实例化了两个委托变量，如下代码：

委托变量dtstatic和dtinstance引用的委托对象的初始状态如下图：



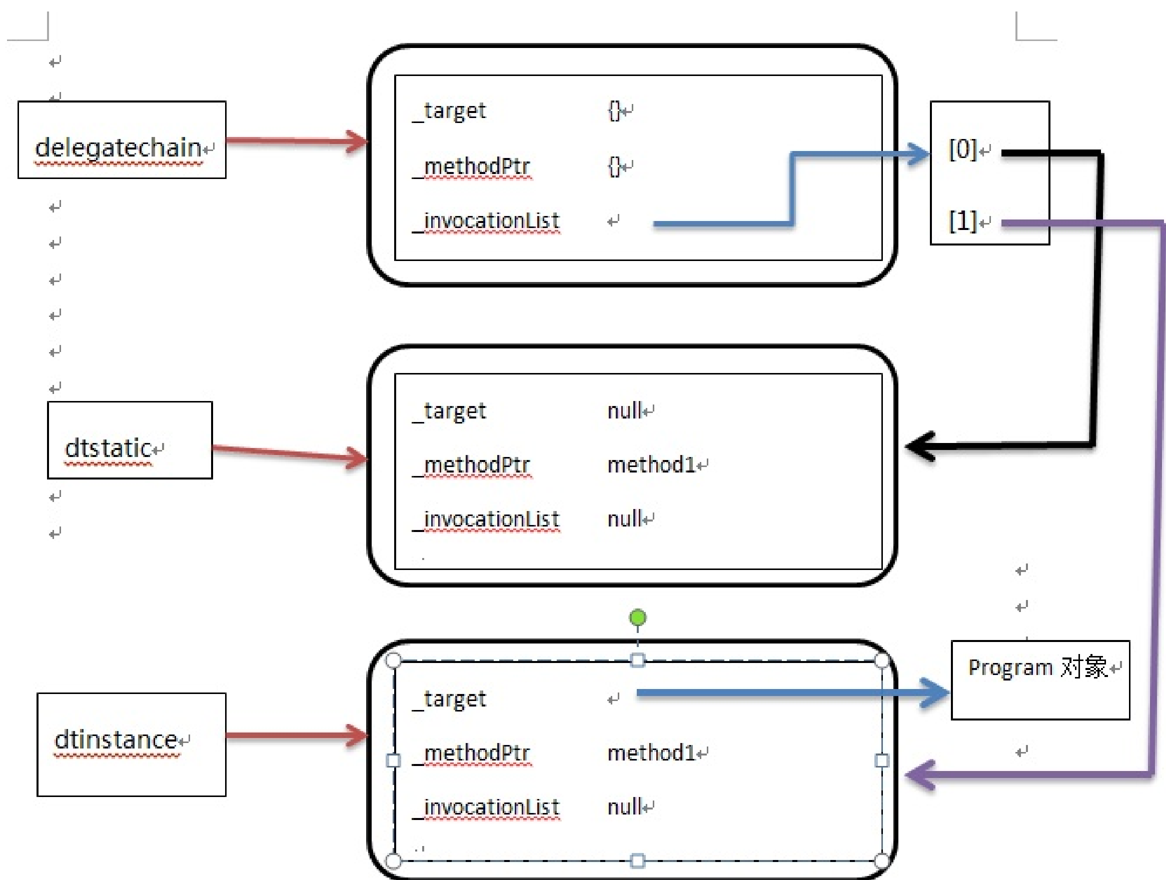
然后我们定义了一个委托类型的引用变量**delegatechain**,刚开始它没有任何委托对象，是一个空引用，当我们执行下面的一行代码时，

Combine方法发现试图合并的是null和dtstatic，在内部，Combine直接返回dtstatic中的对象，此时delegatechain和dtstatic变量引用的都是同一个委托对象，如下图所示：



为了演示委托链，我们通过代码在再添加一个委托，此时就再调用了**Combine**方法，代码如下：

这时候，**Combine**方法发现**delegatechain**已经引用了一个委托对象了（此时已经引用了**destatic**引用的委托对象了），所以**Combine**会构造一个新的委托对象（这一点很想String.Concat,我们简单的使用是通过+操作符把两个字符串连接起来，关于字符串的讨论大家可以参考我博客中的这篇文章http://www.cnblogs.com/zhili/archive/2012/06/25/String_StringBuilder.html），这个新的委托对象会对它的私有字段_target 和_methodPtr字段进行初始化，然后此时**_invocationList**字段初始化为引用了一个委托对象的数组，这个数组的第一个元素（下标为**0**）就是被初始化为引用包装了**method1**方法的委托，数组的二个元素被初始化为引用包装了**method2**方法的委托（也就是**dtinstance**引用的委托对象），最后**delegaechain**被设为引用新建的这个委托对象，下面是一个图，可以帮助大家理解委托链（也叫多播委托）：



同样的道理，如果是添加第三个委托给委托链，过程也是和上面一样的， 此时又会新建一个委托对象，此时_invocationList字段会初始化为引用一个保存这三个委托对象数组，然而有人会问了——对于已经引用了委托对象的委托类型变量调用Combine方法后会创建一个新的委托对象，然后对新的这个委托对象的三个字段进行重新初始化话，最后把之前的委托类型变量引用新创建的委托对象（这里就帮大家总结下委托链的创建过程），那之前的委托对象怎么办呢？ 相信大部分人会有这个疑问的，这点和字符串的Concat方法很像，之前的委托对象和——invocationList字段引用的数组会被垃圾回收掉（正是因为这样，委托和字符串String一样是不可变的）。

注意：我们还可以调用Delegate的Remove方法从链中删除委托，如调用下面代码时：

Remove方法被调用时，它会扫描delegateChain(第一个参数)所引用的委托对象内部维护的委托数组（如果对于委托数组为空的情况下调用Remove方法将不会有任何作用，就是不会删除任何委托引用，这里主要是说明扫描是从委托数组里进行扫描），如果找到delegateChain引用的委托对象的_target和_methodPtr字段

和第二个参数（新创建的委托）中的字段匹配的委托，如果删除之后数组中只剩下一个数据项时，就返回那个数据项（而不会去新建一个委托对象再初始化的，此时的_invocationList为null,而不是保存一个委托对象引用的数组了，具体可以Remove一个后调试看看的），如果此时数组中还剩余多个数据项，就新建一个委托对象——其中创建并初始化_invocationList数组（此时的数组引用的委托对象已经少了一个了，因为用Remove方法删除了），并且，每次**Remove**方法调用只能从链中删除一个委托，而不会删除有匹配的**_target**和**_methodPtr**字段的所有委托（这个大家可以调试看看的）

二、如何对委托链中的委托调用进行控制

通过上面相信大家可以理解如何创建一个委托链对象的，但是从运行结果中还可以看出，每次调用委托链时，委托链包装的每个方法都会顺序被执行，如果委托链中被调用的委托抛出一个异常，这样链中的后续所有对象都不能被调用，并且如果委托的前面具有一个非void的返回类型，则只有最后一个返回值会被保留，其他所有回调方法的返回值都会被舍弃，这就意味着其他所有操作的返回值都永远看不到的吗？事实却不是这样的，我们可以通过调用**Delegate.GetInvocationList**方法来显式调用链中的每一个委托，同时可以添加一些自己的定义输出。

GetInvocationList方法返回一个由Delegate引用构成的数组，其中每一个数组都指向链中的一个委托对象。在内部，**GetInvocationList**创建并初始化一个数组，让数据的每一个元素都引用链中的一个委托，然后返回对该数组的一个引用。如果**_invocatinList**字段为null,返回的数组只有一个元素，该元素就是委托实例本身。下面就通过一个程序来演示下的：

```
namespace DelegateChainDemo
{
    class Program
    {
        // 声明一个委托类型，它的实例引用一个方法
        // 该方法回去一个int 参数，返回void类型
        public delegate string DelegateTest();

        static void Main(string[] args)
        {
            // 用静态方法来实例化委托
            DelegateTest dtstatic = new DelegateTest(Program.methoc

            // 用实例方法来实例化委托
            DelegateTest dtinstance = new DelegateTest(new Program(
            DelegateTest dtinstance2 = new DelegateTest(new Prograr
            // 定义一个委托链对象，一开始初始化为null，就是不代表任何方法（我
            DelegateTest delegatechain = null;
            delegatechain += dtstatic;
            delegatechain += dtinstance;
            delegatechain += dtinstance2;

            ////delegatechain =(DelegateTest)Delegate.Remove(delega
            ////delegatechain = (DelegateTest)Delegate.Remove(deleg
            Console.WriteLine(Test(delegatechain));
            Console.Read();
        }

        private static string method1()
        {
            return "这是静态方法1";
        }

        private string method2()
        {
```

```
                throw new Exception("抛出了一个异常");
            }

            private string method3()
            {
                return "这是实例方法3";
            }
            // 测试调用委托的方法
            private static string Test(DelegateTest chain)
            {
                if (chain == null)
                {
                    return null;
                }

                // 用这个变量来保存输出的字符串
                StringBuilder returnstring = new StringBuilder();

                // 获取一个委托数组，其中每个元素都引用链中的委托
                Delegate[] delegatearray=chain.GetInvocationList();

                // 遍历数组中的每个委托
                foreach (DelegateTest t in delegatearray)
                {
                    try
                    {
                        //调用委托获得返回值
                        returnstring.Append(t() + Environment.NewLine);
                    }
                    catch (Exception e)
                    {
                        returnstring.AppendFormat("异常从 {0} 方法中抛出,
                    }
                }

                // 把结果返回给调用者
                return returnstring.ToString();
            }
        }
    }
```

运行结果截图：

file:///c:/users/administrator/documents/visual studio 2010/Projects/DelegateChainDemo/...

```
这是静态方法1
异常从 method2 方法中抛出，异常信息为：抛出了一个异常
这是实例方法3
```

从运行结果可以看出，此时我们可以获得每一个回调方法的返回值，并且可以加入一些自定义的返回值的（程序中加入了换行字符串），这样就可以对委托链中的每个委托对象进行控制了，即使其中一个抛出异常，此时我们也可以进行捕获，而不会导致后续的委托对象不能被调用的问题。

三、总结下

本专题主要介绍如何创建一个委托链以及对于创建一个委托链的过程进行了详细的分享，第二部分主要先指出了委托了一些局限性，然后通过调用GetInvocationList方法来返回一个委托数组，这样就可以通过遍历委托数组中的每个委托来通知委托的调用过程，这样就可以对委托链的调用进行更多的控制的。到此本专题也就介绍完了，通过这三个专题对委托的介绍，相信大家会对委托有一个更深的理解，然后为什么要写三个专题来详细介绍委托的呢？主要是后面要介绍的事件，Lambda表达式，Linq方面的内容都是和委托有关系的，所以更好的理解委托将是后面特性的一个基础，希望这些对大家有帮助，我将在下一个专题里面介绍事件。

在这里希望大家多多支持下我的IT博客大赛的，我的参赛主页是：http://blog.51cto.com/contest2012/6146675，希望大家帮忙投个票的，谢谢大家的支持

[C# 基础知识系列]专题四：事件揭秘

引言：

前面几个专题对委托进行了详细的介绍的，然后我们在编写代码过程中经常会听到“事件”这个概念的，尤其是写UI的时候，当我们点击一个按钮后VS就会自动帮我们生成一些后台的代码，然后我们就只需要在Click方法里面写代码就可以，所以可能有些刚接触C#的朋友就觉得这样很理所当然的，也没有去思考这是为什么的，为什么点击下事件就会触发我们在Click方法里面写的代码呢？事件到底扮演个什么样的角色呢？为了解除大家的这些疑惑，下面就详细介绍了事件，让一些初学者深入理解C#中的事件的概念。

一、为什么**C#**中会有事件的？

前面专题中介绍了我理解的为什么需要委托的，所以这里我来分享下我理解的为什么C#中要引入事件这个概念的。下面就简单讲讲生活中事件的例子的，最近我生日刚过完的，我就以生日这个话题要谈谈的，日子一天天的过去，当生日的日期到的时候，这时候就触发了生日事件的，此时过生日的人就是触发生日事件的对象的，然后有些关系你的朋友就会对这个事件进行关注，一旦这个事件触发， 他们就可能会陪你一起庆祝生日，然后送礼物给你，当然并不是所有人都会对你的生日关注的，有些人肯定根本就不知道的， 只有对于你生日事件进行了关注了的人才会送礼物给你。这样的生活中的一个生日过程，然而对于为什么C#中会有事件这个概念当然就更好理解了，C#是一个面向对象的语言，我们使用C#语言进行编码也是为了用代码帮助我们完成现实生活中的事情的，所以当然也就必须有事件来反映生活中发生事情的情况了。

二、自己如何实现一个事件模式的？

现在我们知道了为什么C#要引入事件了，但是对于我们在代码中使用的事件大部分都是.net类库为我们提供的，例如控件的各种事件，我们只需要点击按钮后就会触发点击事件的，但是我们很想理解这个事件是如何触发的，我们是否可以自己定义实现事件模式的一个程序的呢？答案当然是可以的，下面就以上面生日的例子来通过代码来解释下如何实现一个事件模式的。

具体代码为：

```
using System;
using System.Threading;

namespace BirthdayEventDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            // 实例化一个事件源对象
            Me eventSource = new Me("Learning Hard");
```

```
            // 实例化关注事件的对象
            Friend1 obj1 = new Friend1();
            Friend2 obj2 = new Friend2();

            // 使用委托把对象及其方法注册到事件中
            eventSource.BirthDayEvent+=new BirthDayEventHandle(obj1
            eventSource.BirthDayEvent+=new BirthDayEventHandle(obj2

            // 事件到了触发生日事件，事件的调用
            eventSource.TimeUp();
            Console.Read();
        }
    }
    // 第一步： 定义一个类型用来保存所有需要发送给事件接收者的附加信息
    public class BirthdayEventArgs : EventArgs
    {
        // 表示过生日人的姓名
        private readonly string name;

        public string Name
        {
            get { return name;}
        }

        public BirthdayEventArgs(string name)
        {
            this.name = name;
        }
    }
   // 第二步：定义一个生日事件，首先需要定义一个委托类型，用于指定事件触发时被
    public delegate void BirthDayEventHandle(object sender, Birthda
    // 定义事件成员
    public class Subject
    {
        // 定义生日事件
        public event BirthDayEventHandle BirthDayEvent;

        // 第三步：定义一个负责引发事件的方法，它通知已关注的对象（通知我的好
        protected virtual void Notify(BirthdayEventArgs e)
        {
            // 出于线程安全的考虑，现在将对委托字段的引用复制到一个临时字段中
            BirthDayEventHandle temp = Interlocked.CompareExchange(
            if (temp != null)
            {
                // 触发事件，与方法的使用方式相同
                // 事件通知委托对象，委托对象调用封装的方法
                temp(this, e);
            }
        }
    }

    // 定义触发事件的对象，事件源
    public class Me : Subject
```

```
    {
        private string name;
        public Me(string name)
        {
            this.name = name;
        }
        public void TimeUp()
        {
            BirthdayEventArgs eventarg = new BirthdayEventArgs(name
            // 生日到了，通知朋友们
            this.Notify(eventarg);

        }
    }

    // 好友对象
    public class Friend1
    {
        public void SendGift(object sender,BirthdayEventArgs e)
        {
            Console.WriteLine(e.Name+" 生日到了，我要送礼物");
        }
    }
    public class Friend2
    {
        public void Buycake(object sender, BirthdayEventArgs e)
        {
            Console.WriteLine(e.Name + " 生日到了,我要准备买蛋糕");
        }
    }
}
```

运行结果为：

```
file:///c:/users/administrator/documents/visual studio 2010/Projects/BirthdayEventDemo/...
Learning Hard 生日到了，我要送礼物
Learning Hard 生日到了,我要准备买蛋糕
```

三、编译器是如何解释事件的呢？

上面我们已经介绍了如何去实现自己去实现一个事件模式的，大家可以展开代码来具体的查看的，实现过程主要是——定义触发对象的事件源（指的是谁过生日）-> 定义关注你生日事件的朋友对象-> 方法登记对事件的关注，当事件触发时通知登记的方法被调用。然而相信大家还有有疑问——到底C#中的事件是什么呢？编译器又是如何去解释它的？下面就为大家解除下疑惑的：

首先事件其实就是委托的（确切的说事件就是委托链），从上面的代码中，我们定义的事件除了使用event关键字外，还用到了一个委托类型，然而委托是一个类，类肯定就有属性字段的，然而我们就可以把事件理解为委托的一个属性，属性的返回值是一个委托类型。说事件是委托的一个属性，是有根据的，我们通过中间语言代码可以知道编译器是如何去解释我们定义的事件的。

当我们像上面定义一个事件时，编译器会把它转换为3段代码（大家可以通过IL反汇编程序来查看的）：

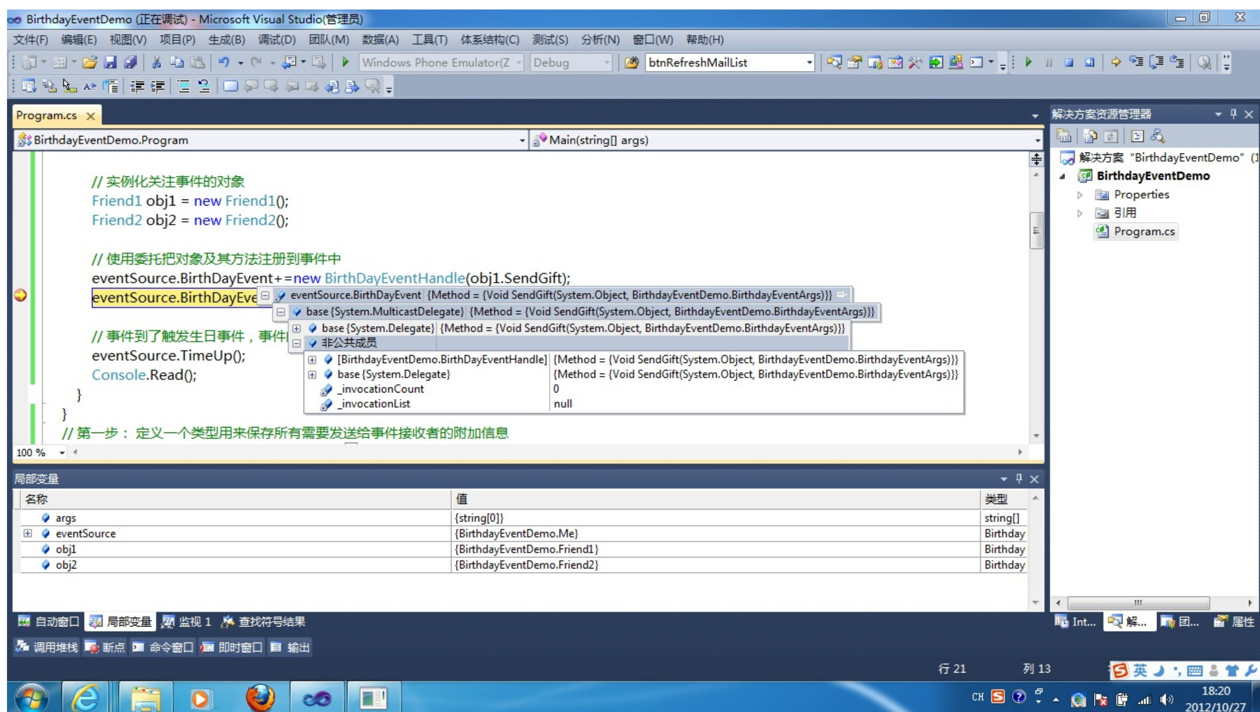
```
      // 1\. 一个被初始化为null的私有委托字段
        private BirthDayEventHandle BirthDayEvent =null;

         //2\. 一个公共add_BirthDayEvent方法
         public void add_BirthDayEvent(BirthDayEventHandle value)
         {
             // 以一种线程安全的方式从事件中添加一个委托
         }
         // 3\. 一个公共的remove_BirthDayEvent方法
         public void remove_BirthDayEvent(BirthDayEventHandle value)
         {
             // 以一种线程安全的方式从事件中移除一个委托
         }
```

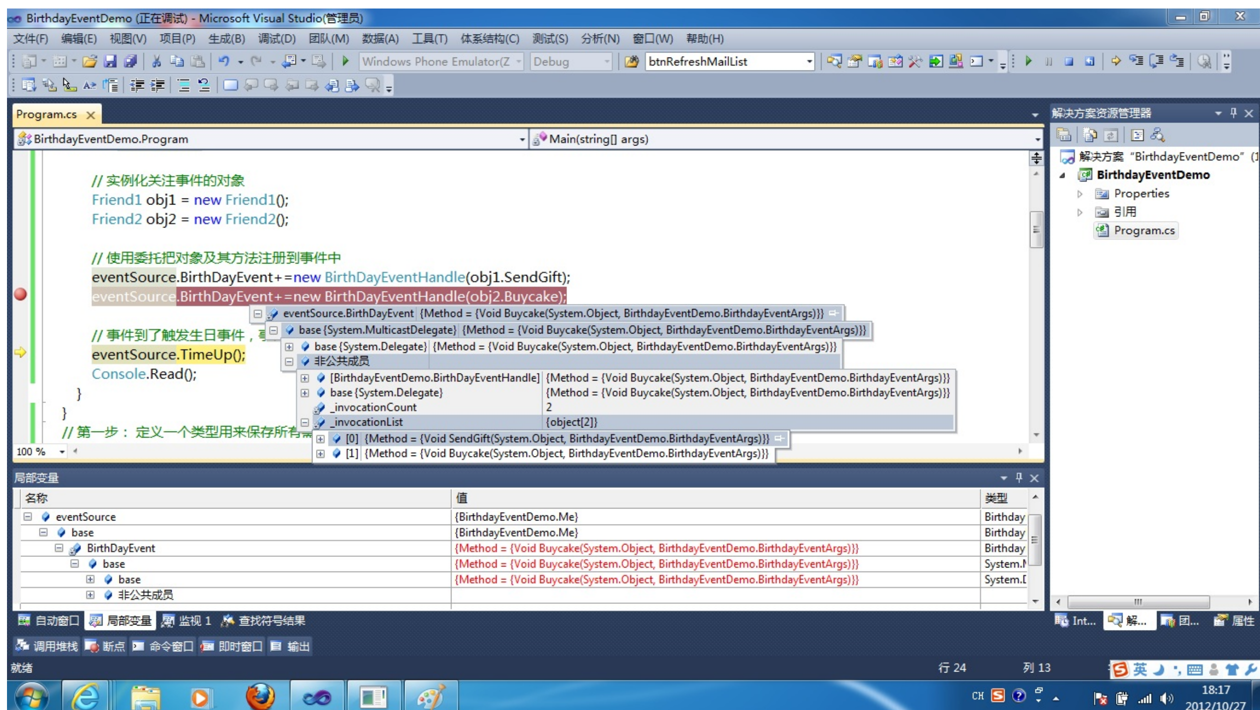
第一段代码一个委托的私有字段，该字段是对一个委托列表的头部的引用，事件发生时会通知这个列表中的委托。字段初始化为null,表明无关注人登记了对事件的关注。 第二段代码是一个以add为前缀的方法，该方法是由编译器自动命名的，代码内容调用**Delegate.Combine**方法将委托实例添加到委托列表中，返回新的列表地址，并将这个地址存回字段。

第三段代码也是一个方法，它使得一个对象注销对事件的关注，同样的方法体调用Delegate.Remove方法将委托实例从委托列表中删除，返回新的列表地址，并将这个地址存回字段中。（注，如果试图删除一个从未添加过的方法，**Delegate.Remove**方法在内部将不做任何事情，也就是说，不会抛出任何一次，也不会显示任何警告，事件的方法集合保持不变）。

同时大家也可以通过调试来说明事件是一个委托链的，大家可以在**eventSource.BirthDayEvent+=new BirthDayEventHandle(obj2.Buycake);**这行代码设置一个断点调试的，下面是我调试过程中的一个截图，大家也可以自己调试看看的，这样将会更加理解事件是一个委托链的概念：



按F10运行一行后的截图



通过上面的截图，相信大家对于事件是一个委托链的概念相信会有进一步的理解的。

四、小结

到这里本专题的内容也就介绍完了， 希望通过本专题，大家可以对事件有进一步的理解，理解事件与委托之间的关系。这个专题通过自己实现的一个事件模式里解释事件的本质，然而我们经常使用的是Net类库中定义好的事件，然而有些刚接触C#的人却不理解Net中定义的事件背后所做的事情，只是知道点下按钮后在Click方法

里面写入自己的一些控制代码，然而背后的过程具体是怎样的，既然事件是委托，那么Click事件是委托类型，其中的委托类型又是怎么被实例化的呢？这些内容将在下一个专题给大家分享下的。

[C# 基础知识系列]专题五：当点击按钮时触发Click事件背后发生的事情

引言：

当我们在点击窗口中的Button控件VS会帮我们自动生成一些代码，我们只需要在Click方法中写一些自己的代码就可以实现触发Click事件后我们Click方法中代码就会执行，然而我一直有一个疑问的——既然上一专题中说事件是一个多播委托，然而自动生成的代码中只有事件的实例化，却没有看到事件的调用，那既然没有事件调用的代码，那封装的Click为什么会执行呢？

一、点击按钮时触发**Click**事件背后发送的事情

在引言中提出了我的提问的， 我相信有些朋友可能也会有这样的疑问的，然后事件肯定是调用了的， 只是不是我们代码中调用，而是Butoon控件的内部代码里面调用了事件，而导致委托封装的Click方法而被调用，这样才符合我们看到的情况的——我们点击按钮后，我们后台代码中的Click方法就会执行。为了明白到底背后发生了什么事情的， 让我们一起来探究个究竟吧？

我们新建一个Windows 窗体程序，然后在窗体中拖入一个Button控件并单击按钮，这时候VS为我们生成了如下的代码：

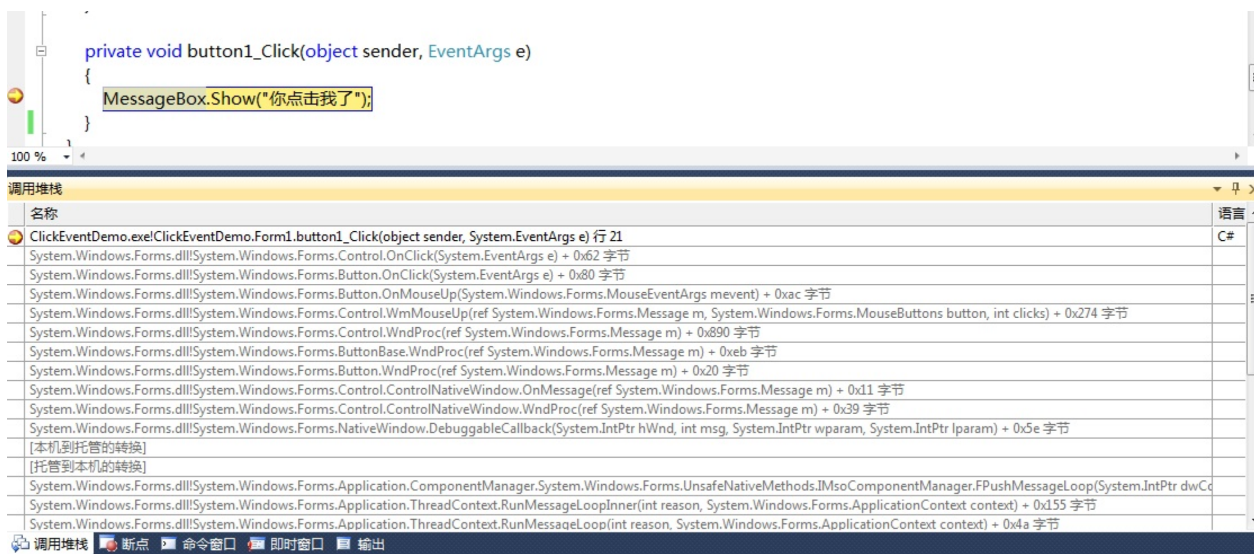
```
private System.Windows.Forms.Button button1;
private void InitializeComponent()
       {
           his.button1 = new System.Windows.Forms.Button();

this.button1.Location = new System.Drawing.Point(105, 89);
           this.button1.Name = "button1";
           this.button1.Size = new System.Drawing.Size(75, 23);
           this.button1.TabIndex = 0;
           this.button1.Text = "请点击我";
           this.button1.UseVisualStyleBackColor = true;
           this.button1.Click += new System.EventHandler(this.butt
}
// 后台代码
 private void button1_Click(object sender, EventArgs e)
       {

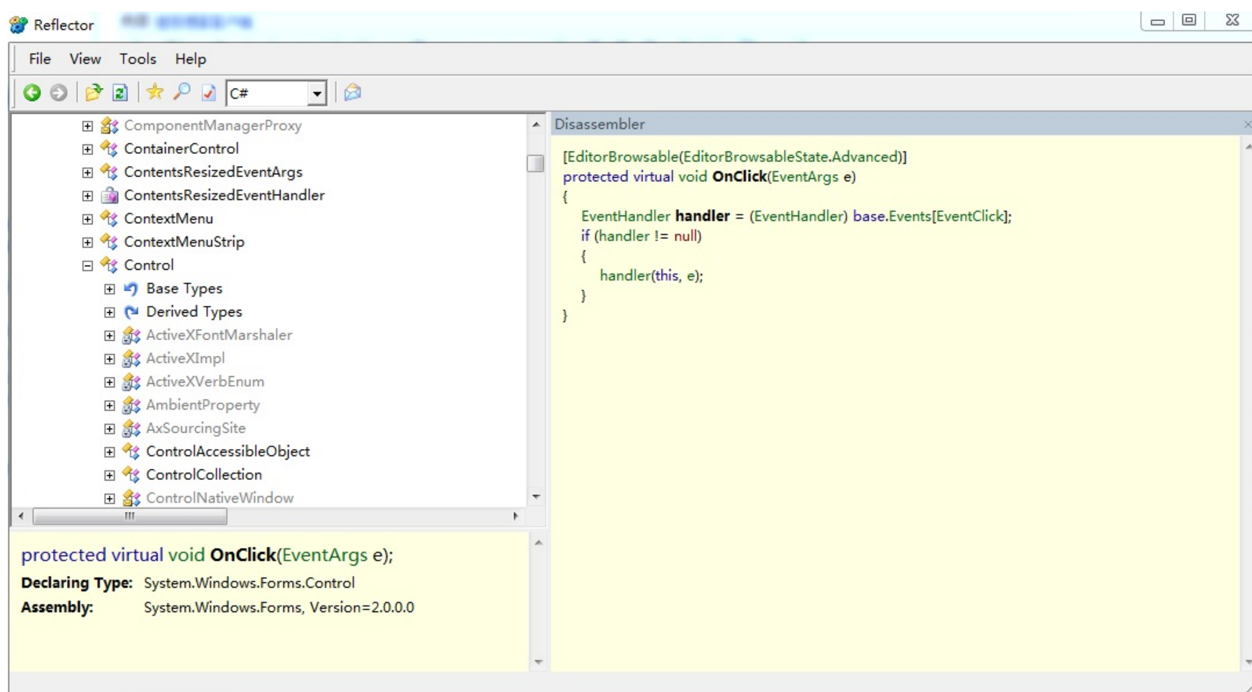
       }
```

从上面代码中我们看到VS为我们自动创建了一个Button对象并实例化，设置了它的属性并通过 **this.button1.Click += new System.EventHandler(this.button1_Click);**这行代码把 **button1_Click**注册对

Click事件的关注，然而事件的调用代码在哪里呢？下面我们就在button1_Click方法里面设置断点看看代码是如何执行的（通过查看调用堆栈来看看代码的执行顺序），下面是我设置断点的一张调用堆栈截图：

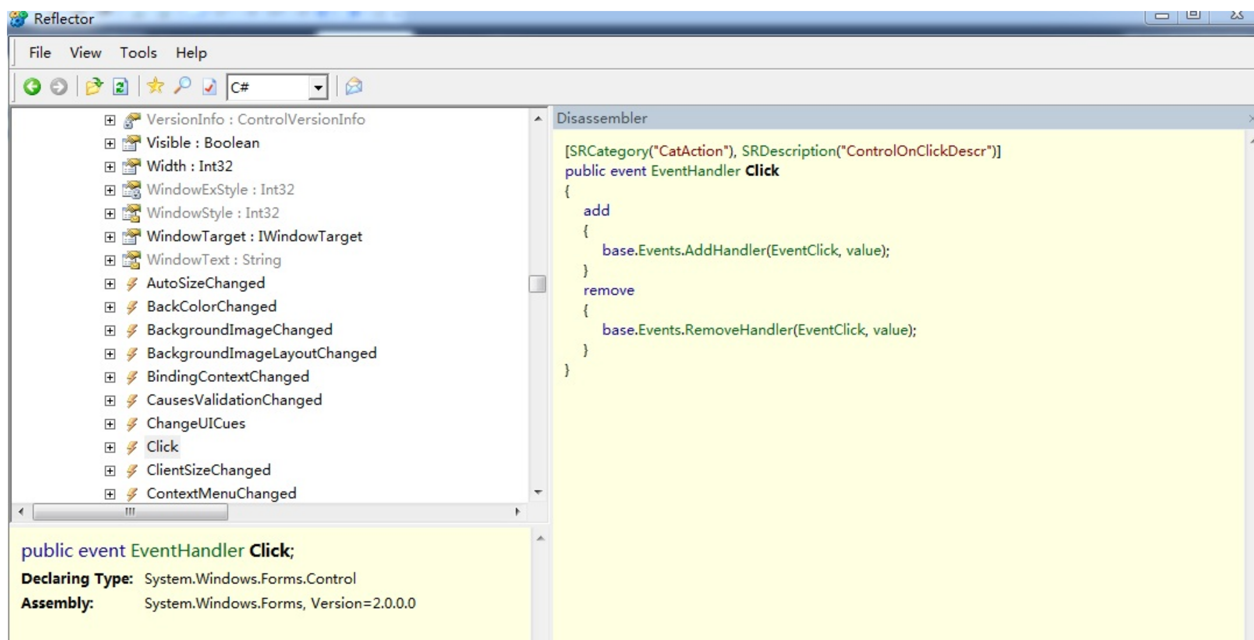


从上图中我发现在调用button1_Click方法之前要执行**Control.OnClick（System.EventArgs e）**方法的，然后我们用发射工具去查看下**Control.OnClick(System.Eventrgs e)**方法中具体有什么样的代码：OnClick方法内部代码截图为：



从反射的代码中可以明白，首先从**Events**（大家可以通过反射工具去查看**Events**的类型，它的类型为**EventHandlerList**,而**EventHandlerList**又是一个密封类）委托集合中取出委托，如果Click事件（委托）实例化了的话，此时就不为空，此时就会调用委托——**handler(this, e)**,我们知道之前我们通过 this.button1.Click += new System.EventHandler(this.button1_Click);代码实例化了委托事件，所以此时被EventHandler封装的button1_Click方法就会执行。**

通过上面的解释我已经解除了我一开始的疑惑了，事件的调用在.Net类库中的**Control.OnClick**方法里面调用，这也就是我说要表达的**Click**事件背后做的事情的下面是反射得到的**Click**事件的代码截图：



二、小结

本专题首先提出我对按钮单击事件背后发生的事情的疑惑，通过调试和反射工具一步一步把疑惑接触，相信其他控件的其他事件也是如此的，本专题主要想让大家知道下.Net类库为我们做的事情的，希望一些初学者们了解知识时，要努力知道事物的本质。最后希望本专题可以让大家更进一步的理解事件的本质的，我将下一专题和大家分享下我理解的泛型到底是怎样的。

反射工具的下载地址：http://files.cnblogs.com/zhili/Reflector.zip

[C# 基础知识系列]专题六:泛型基础篇——为什么引入泛型

引言：

前面专题主要介绍了C#1中的2个核心特性——委托和事件，然而在C# 2.0中又引入一个很重要的特性，它就是泛型，大家在平常的操作中肯定会经常碰到并使用它，如果你对于它的一些相关特性还不是很了解，那就让我们一起进入本专题的学习的。

一、泛型的是什么

泛型的英文解释为generic，当然我们查询这个单词时，更多的解释是通用的意思，然而有些人会认为明明是通用类型，怎么成泛型了的，其实这两者并不冲突的，泛型本来代表的就是通用类型，只是微软可能有一个比较官方的此来形容自己引入的特性而已，既然泛型是通用的， 那么泛型类型就是通用类型的，即泛型就是一中模子。 在生活中，我们经常会看到模子，像我们平常生活中用的桶子就是一个模子，我们可以用桶子装水，也可以用来装油，牛奶等等，然而把这些都装进桶子里面之后，它们都会具有桶的形状（水，牛奶和油本来是没有形的），即具有模子的特征。同样，泛型也是像桶子一样的模子，我们可以用int类型，string类型， 类去实例化泛型，实例化之后int,string类型都会具有泛型类型的特征（就是说可以使用泛型类型中定义的方法，如List<T>泛型，如果用int去初始化它后，List<int>的实例就可以用List<T>泛型中定义的所有方法，用string去初始化它也一样，和我们生活中的用桶装水，牛奶，油等非常类似）

二、**C# 2.0**为什么要引入泛型

大家通过第一部分知道了什么是泛型，然而C#2.0中为什么要引入泛型的？这答案当然是泛型有很多好处的。下面通过一个例子来说明C# 2.0中为什么要引入泛型，然后再介绍下泛型所带来的好处有哪些。

当我们要写一个比较两个整数大小的方法时，我们可能很快会写出下面的代码：

```
public class Compare
   {
       // 返回两个整数中大的那一项
       public static int Compareint(int int1, int int2)
       {
           if (int1.CompareTo(int2) > 0)
           {
               return int1;
           }

           return int2;
       }
   }
```

然而需求改变为又要实现比较两个字符串的大小的方法时，我们又不得不在类中实现一个比较字符串的方法：

```
  // 返回两个字符串中大的一项
        public static string Comparestring(string str1, string str2
        {
            if (str1.CompareTo(str2) > 0)
            {
                return str1;
            }

            return str2;
        }
```

如果需求又改为要实现比较两个对象之间的大小时，这时候我们又得实现比较两个对象大小的方法，然而我们中需求中可以看出，需求中只是比较的类型不一样的，其实现方式是完全一样的，这时候我们就想有没有一种类型是通用的，我们可以把任何类型当做参数传入到这个类型中去实例化为具体类型的比较，正是有了这个想法，同时微软在C#2.0中也想到了这个问题，所以就导致了C#2.0中添加了泛型这个新的特性，泛型就是——通用类型，有了泛型之后就可以很好的帮助我们刚才遇到的问题的，这样就解决了我们的第一个疑问——为什么要引入泛型。下面是泛型的实现方法：

```
public class Compare<T> where T : IComparable
    {
        public  static T CompareGeneric(T t1, T t2)
        {
            if (t1.CompareTo(t2) > 0)
            {
                return t1;
            }
            else
            {
                return t2;
            }
        }
    }
```

这样我们就不需要针对每个类型实现一个比较方法，我们可以通过下面的方式在主函数中进行调用的：

```
public class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(Compare<int>.CompareGeneric(3, 4));
            Console.WriteLine(Compare<string>.CompareGeneric("abc",
            Console.Read();
        }
    }
```

通过上面的代码大家肯定可以理解C# 2.0中为什么要引入泛型的，然而泛型可以给我们带什么好处的呢？从上面的例子可以看出，泛型可以帮助我们实现代码的重用，大家很清楚——面向对象中的继承也可以实现代码的重用，然而泛型提供的代码的重用，确切的说应该是 "算法的重用"（我理解的算法的重用是我们在实现一个方法中，我们只要去考虑如何去实现算法，而不需要考虑算法操作的数据类型的不同，这样的算法实现更好的重用，泛型就是提供这样的一个机制）。

然而泛型除了实现代码的重用的好处外，还有可以提供更好的性能和类型安全，下面通过下面一段代码来解释下为什么有这两个好处的。

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Diagnostics;

namespace GeneralDemo
{
    public class Program
    {
        static void Main(string[] args)
        {
            Stopwatch stopwatch = new Stopwatch();

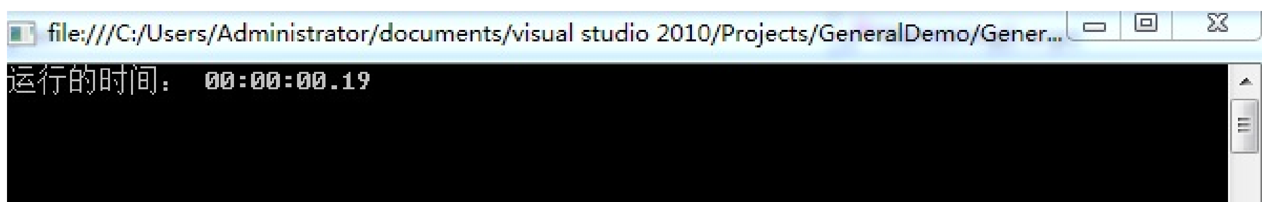
            // 非泛型数组
            ArrayList arraylist = new ArrayList();

            // 泛型数组
            List<int> genericlist= new List<int>();
            // 开始计时
            stopwatch.Start();
            for (int i = 1; i < 10000000; i++)
            {
                genericlist.Add(i);
                ////arraylist.Add(i);
            }

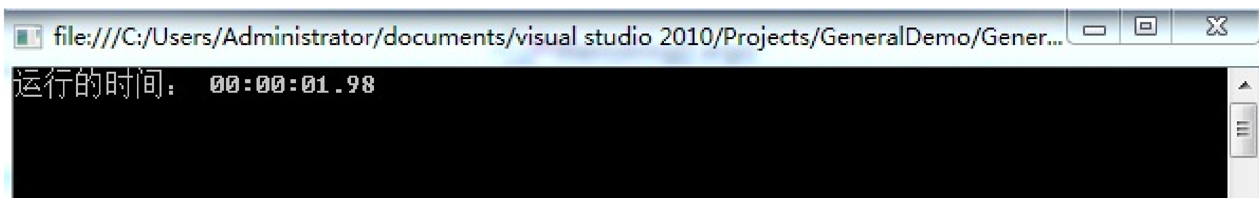
            // 结束计时
            stopwatch.Stop();

            // 输出所用的时间
            TimeSpan ts = stopwatch.Elapsed;
            string elapsedTime = String.Format("{0:00}:{1:00}:{2:00
             ts.Hours, ts.Minutes, ts.Seconds,
             ts.Milliseconds/10);
            Console.WriteLine("运行的时间： " + elapsedTime);
            Console.Read();
        }
    }
}
```

当我们把 arraylist.Add(i);这行代码注释掉来测试向泛型数组中加入数据的运行时间，下面是我机器的上运行的一个截图：



当我们把 genericlist.Add(i);这行代码注释掉来测试向一个非泛型数组中加入数据的运行时间，下面附上我机器上的运行的时间截图：



从两个结果中就可以明显看出 向泛型数组中的加入数据的效率远高于非泛型数组。有图有真相，这样就充分说明泛型的另一个好处——高性能，然而泛型类型也保证了类型安全（大家都知道，C#是一个强类型的语言的，**强**类型指的是在每定义一个变量都需要指定变量的类型），当我们向这个泛型genericlist数组中添加string类型的值时，此时就会造成编译器报错 “无法从**“string”**转换为**’int‘** ”

三、小结

本专题主要和大家分享了C# 2.0中为什么会引入委托，以及委托的好处，相信通过上面的介绍大家可以对委托有一个简单的认识以及对于泛型所带来的好处也有一个全面的认识，对于泛型的高性能本专题并没有给出原因，这个内容将会在下面一个专题向大家介绍。

[C# 基础知识系列]专题七: 泛型深入理解(一)

引言：

在上一个专题中介绍了C#2.0 中引入泛型的原因以及有了泛型后所带来的好处，然而上一专题相当于是介绍了泛型的一些基本知识的，对于泛型的性能为什么会比非泛型的性能高却没有给出理由，所以在这个专题就中将会介绍原因和一些关于泛型的其他知识。

一、泛型类型和类型参数

泛型类型和其他int,string一样都是一种类型，泛型类型有两种表现形式的：泛型类型（包括类、接口、委托和结构，但是没有泛型枚举的）和泛型方法。那什么样的类、接口、委托和方法才称作泛型类型的呢 ？我的理解是类、接口、委托、结构或方法中有类型参数就是泛型类型，这样就有类型参数的概念的。 类型参数 ——是一个真实类型的一个占位符（我想到一个很形象的比喻的，比如大家在学校的时候，一到中午下课的时候食堂人特别多的，所以很多应该都有用书本占位置的习惯的， 书本就相当于一个占位符，真真坐在位置上的当然是自己的，讲到占位置，以前听过我同学说，他们班有个很牛逼的**MM**，中午下完课的时候用手机占位子的，等它打完饭回来的时候手机已经不见， 当时听完我就和我同学说，你们班这位女生真牛逼的，后面我们就🤔），泛型声明中，类型参数必须放在一对尖括号里面（即<>这个符号），并且用逗号分隔多个类型参数，如List<T>类中T就是类型参数，在使用泛型类型或方法的时候，我们要用真实类型来代替，就像用书本占位子一个，书本只是暂时的在那个位置上，等打好饭了就要换成你坐在位置上了，同样在C#中泛型也是同样道理，类型参数只是暂时的在那个位置，真真使用中要用真实的类型去代替它的位置，此时我们把真实类型又取名为类型实参，如上一专题的代码中List<int>，类型实参就是int(代替T的位置)。

如果没有为类型参数提供类型实参，此时我们就声明了一个未绑定的泛型类型，如果指定了类型实参，此时的类型就叫做已构造类型（这里同样可以以书占位置去理解），然而已构造类型又可以是开放类型或封闭类型的，这里先给出这个两个概念的定义的：开放类型——具有类型参数的类型就是开放类型（所有的未绑定的泛型类型都属于开放类型的），封闭类型——为每个类型参数都传递了实际的数据类型。对于开放类型，我们创建开放类型的实例。

注意*:*在*C#*代码中，我们唯一可以看到未绑定泛型类型的地方（除了作为声明之外）就是在*typeof*操作符里。

下面通过以下代码来更好的说明这点：

```
using System;
using System.Collections.Generic;

namespace CloseTypeAndOpenType
{
    // 声明开放泛型类型
    public sealed class DictionaryStringKey<T> : Dictionary<string,
```

```
    {

    }

   public class Program
    {
        static void Main(string[] args)
        {
            object o = null;

            // Dictionary<,>是一个开放类型，它有2个类型参数
            Type t = typeof(Dictionary<,>);

            // 创建开放类型的实例（创建失败，出现异常）
            o = CreateInstance(t);
            Console.WriteLine();

            // DictionaryStringKey<>也是一个开放类型，但它有1个类型参数
            t = typeof(DictionaryStringKey<>);

            // 创建该类型的实例（同样会失败，出现异常）
            o = CreateInstance(t);
            Console.WriteLine();

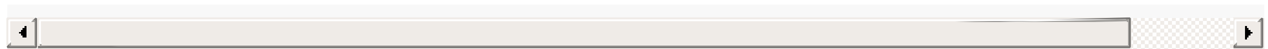
            // DictionaryStringKey<int>是一个封闭类型
            t = typeof(DictionaryStringKey<int>);

            // 创建封闭类型的一个实例（成功）
            o = CreateInstance(t);

            Console.WriteLine("对象类型 = " + o.GetType());
            Console.Read();
        }

       // 创建类型
        private static object CreateInstance(Type t)
        {
            object o = null;
            try
            {
                // 使用指定类型t的默认构造函数来创建该类型的实例
                o = Activator.CreateInstance(t);
                Console.WriteLine("已创建{0}的实例", t.ToString());
            }
            catch(Exception ex)
            {
                Console.WriteLine(ex.Message);
            }

            return o;
        }
    }
}
```



运行结果为（从结果中也可以看出开放类型不能创建该类型的一个实例，异常信息中指出类型中包含泛型参数）：

```
file:///C:/Users/Administrator/documents/visual studio 2010/Projects/GeneralDemo/Close...
无法创建 System.Collections.Generic.Dictionary`2[TKey,TValue] 的实例，因为 Type.ContainsGenericParameters 为 True。

无法创建 CloseTypeAndOpenType.DictionaryStringKey`1[T] 的实例，因为 Type.ContainsGenericParameters 为 True。

已创建CloseTypeAndOpenType.DictionaryStringKey`1[System.Int32]的实例
对象类型 = CloseTypeAndOpenType.DictionaryStringKey`1[System.Int32]
```

二、泛型类型中的静态字段和静态构造函数

首先实例字段是属于一个实例的，静态字段是从属于它们声明的类型，即如果在某个Myclass类中声明了一个静态字段field,则不管创建Myclass的多少个实例，也不管从Myclass中派生出多少个实例，都只有一个Myclass.x字段。然而每个封闭类型都有它自己的静态字段（使用类型实参时，实际上**CLR**会定义一个新的类型对象，所以每个静态字段都是不一样对象里面的静态字段，所以才会每个都有各自的值）通过以下代码来更好说明下——每个封闭类型都有它自己的静态字段：

View Code

```
namespace GenericStaticFieldAndStaticFunction
{
    // 泛型类，具有一个类型参数
    public static class TypeWithStaticField<T>
    {
        public static string field;
        public static void OutField()
        {
            Console.WriteLine(field+":"+typeof(T).Name);
        }
    }

    // 非泛型类
    public static class NoGenericTypeWithStaticField
    {
        public static string field;
        public static void OutField()
        {
            Console.WriteLine(field);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            // 使用类型实参时，实际上CLR会定义一个新的类型对象
            // 所以每个静态字段都是不一样对象里面的静态字段，所以才会每个都有
            // 对泛型类型类的静态字段赋值
            TypeWithStaticField<int>.field = "一";
            TypeWithStaticField<string>.field = "二";
            TypeWithStaticField<Guid>.field = "三";

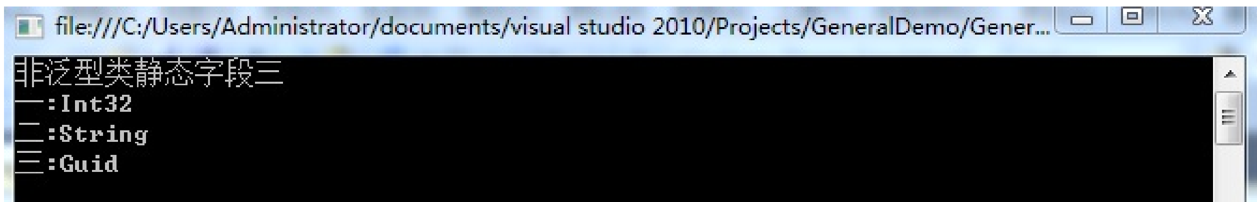
            // 此时filed 值只会有一个值，每个赋值都是改变了原来的值
            NoGenericTypeWithStaticField.field = "非泛型类静态字段一"
            NoGenericTypeWithStaticField.field = "非泛型类静态字段二"
            NoGenericTypeWithStaticField.field = "非泛型类静态字段三"

            NoGenericTypeWithStaticField.OutField();

            // 证明每个封闭类型都有一个静态字段
            TypeWithStaticField<int>.OutField();
            TypeWithStaticField<string>.OutField();
            TypeWithStaticField<Guid>.OutField();
            Console.Read();

        }
    }
}
```

运行结果：



同样每个封闭类型都有一个静态构造函数的，通过下面的代码可以让大家更加明白这点：

```
    // 静态构造函数的例子
     public static class Outer<Tx>
     {
         // 嵌套类
         public class Inner<Ty>
         {
             // 静态构造函数
             static Inner()
             {
                 Console.WriteLine("Outer<{0}>.Inner<{1}>", typeof(T
             }

             public static void Print()
             {
             }
         }
     }
 class Program
     {
         static void Main(string[] args)
         {
    #region 静态函数的演示

             // 静态构造函数会运行多次
             // 因为每个封闭类型都有单独的一个静态构造函数
             Outer<int>.Inner<string>.Print();
             Outer<int>.Inner<int>.Print();
             Outer<string>.Inner<int>.Print();
             Outer<string>.Inner<string>.Print();
             Outer<object>.Inner<string>.Print();
             Outer<object>.Inner<object>.Print();
             Outer<string>.Inner<int>.Print();
             Console.Read();
             #endregion
         }
     }
```

运行结果：

file:///C:/Users/Administrator/documents/visual studio 2010/Projects/GeneralDemo/Gener...

```
Outer<System.Int32>.Inner<System.String>
Outer<System.Int32>.Inner<System.Int32>
Outer<System.String>.Inner<System.Int32>
Outer<System.String>.Inner<System.String>
Outer<System.Object>.Inner<System.String>
Outer<System.Object>.Inner<System.Object>
```

从上图的运行结果可能会发现，我们代码中7个需要输出的，但是结果中只有6个结果输出的，这是因为任何封闭类型的静态构造函数只执行一次，最后一行的Outer<string>.Inner<int>.Print();这行不会产生第7行输出， 因为Outer<string>.Inner<int>.Print();的静态构造函数在之前已经执行过的（第三行已经执行过了）。

三、编译器如何解析泛型

在上一个专题中，我只是贴出了泛型与非泛型的比较结果来说明泛型具有高性能的好处，却没有给出具体导致泛型比非泛型效率高的原因，所以在这个部分来剖析下泛型效率的具体原因。

这里先贴出上一个专题中说明泛型高性能好处的代码，然后再查看IL代码来说明泛型的高性能（针对泛型和非泛型，C#编译器是如何解析为IL代码的）：

View Code

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Diagnostics;

namespace GeneralDemo
{
    public class Program
    {
        static void Main(string[] args)
        {
            Stopwatch stopwatch = new Stopwatch();

            // 非泛型数组
            ArrayList arraylist = new ArrayList();

            // 泛型数组
            List<int> genericlist= new List<int>();

            // 开始计时
            stopwatch.Start();
            for (int i = 1; i < 10000000; i++)
            {
                //genericlist.Add(i);
                arraylist.Add(i);
            }

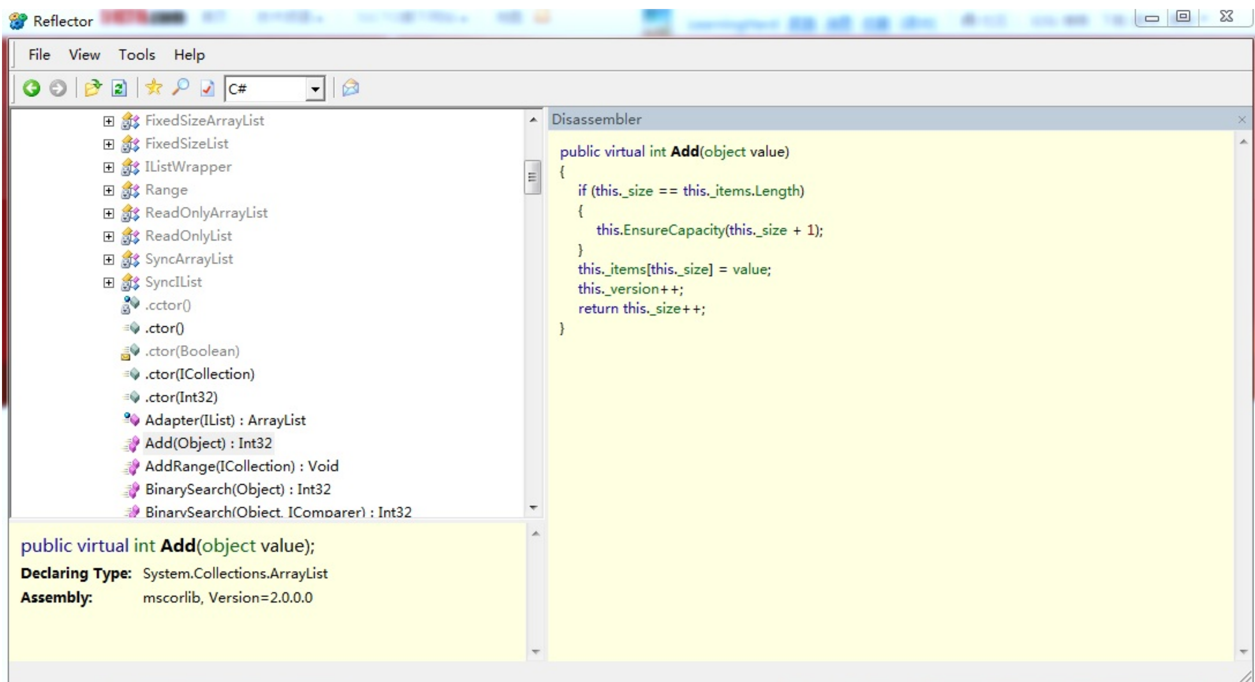
            // 结束计时
            stopwatch.Stop();

            // 输出所用的时间
            TimeSpan ts = stopwatch.Elapsed;
            string elapsedTime = String.Format("{0:00}:{1:00}:{2:0(
             ts.Hours, ts.Minutes, ts.Seconds,
             ts.Milliseconds/10);
            Console.WriteLine("运行的时间： " + elapsedTime);
            Console.Read();
        }
    }
}
```

当使用用非泛型的的ArrayList数组时，IL的代码如下（这里只是贴出了部分主要的中间代码，具体的大家可以下载示例源码用IL反汇编程序查看的）：

```
  IL_001f:  ldloc.1
  IL_0020:  ldloc.3
  IL_0021:  **box       [mscorlib]System.Int32
**  IL_0026:  callvirt   instance int32 [mscorlib]System.Collection
  IL_002b:  pop
  IL_002c:  nop
  IL_002d:  ldloc.3
  IL_002e:  ldc.i4.1
  IL_002f:  add
```

在上面的IL代码中，我用红色的标记的代码主要是在执行装箱操作（装箱过程肯定是要消耗的事件的吧， 就像生活中寄包裹一样，包装起来肯定是要花费一定的时间的， 装箱操作同样会，然而对于泛型类型就可以避免装箱操作，下面会贴出使用泛型类型的IL代码的截图）——这个操作也是影响非泛型的性能不如泛型类型的根本原因。然而为什么使用**ArrayList**类型在调用Add方法来向数组添加元素之前要装箱的呢？原因其实主要出在Add方法上的， 大家可以用**Reflector**反射工具查看ArrayList的Add方法定义，下面是一张Add方法原型的截图：



从上面截图可以看出，Add(objec value)需要接收object类型的参数，然而我们代码中需要传递的是int实参，此时就需要会发生装箱操作（值类型int转化为object引用类型,这个过程就是装箱操作）， 这样也就解释了为什么调用Add方法会执行装箱操作的， 同时也就说明泛型的高性能的好处。

下面是使用泛型**List<T>**的**IL**代码截图（从图片中可以看出，使用泛型时，没有执行装箱的操作， 这样就少了装箱的时间， 这样当然就运行的快了， 性能就好了。）：

GeneralDemo.Program::Main : void(string[])

查找(F)　查找下一个(N)

```
IL_001b:  stloc.3
IL_001c:  br.s       IL_002c
IL_001e:  nop
IL_001f:  ldloc.2
IL_0020:  ldloc.3
IL_0021:  callvirt   instance void class [mscorlib]System.Collections.Generic.List`1<int32>::Add(!0)
IL_0026:  nop
IL_0027:  nop
IL_0028:  ldloc.3
IL_0029:  ldc.i4.1
IL_002a:  add
```

四、小结

说到这里本专题的内容也就介绍结束了，本专题主要是进一步介绍了泛型的其他内容的，由于篇幅的关于我将泛型的其他内容放在下一专题中，如果都在放在这个专题中内容会显得非常多，这样也不利于大家的消化和大家的阅读，所以我在下一个专题中继续介绍泛型的其他的一些内容。

下面先附上泛型专题中用到的所有Demo的源代码：http://files.cnblogs.com/zhili/GeneralDemo.zip

[C# 基础知识系列]专题八: 深入理解泛型(二)

引言：

本专题主要是承接上一个专题要继续介绍泛型的其他内容，这里就不多说了，就直接进入本专题的内容的。

一、类型推断

在我们写泛型代码的时候经常有大量的"<"和">"符号，这样有时候代码一多，也难免会让开发者在阅读代码过程中会觉得有点晕的，此时我们觉得晕的时候肯定就会这样想：是不是能够省掉一些"<" 和">"符号的呢？你有这种需求了， 当然微软这位好人肯定也会帮你解决问题的，这样就有了我们这部分的内容——类型推断，意味着编译器会在调用一个泛型方法时自动判断要使用的类型，（这里要注意的是：类型推断只使用于泛型方法，不适用于泛型类型），下面是演示代码：

```
using System;

namespace 类型推断例子
{
    class Program
    {
        static void Main(string[] args)
        {
            int n1 = 1;
            int n2 = 2;
            // 没有类型推断时需要写的代码
            // GenericMethodTest<int>(ref n1, ref n2);

            // 有了类型推断后需要写的代码
            // 此时编译器可以根据传递的实参 1和2来判断应该使用Int类型实参来
            // 可以看出有了类型推断之后少了<>,这样代码多的时候可以增强可读性
            GenericMethodTest(ref n1, ref n2);
            Console.WriteLine("n1的值现在为：" + n1);
            Console.WriteLine("n2的值现在为：" + n2);
            Console.Read();

            //string t1 = "123";
            //object t2 = "456";
            //// 此时编译出错，不能推断类型
            //// 使用类型推断时，C#使用变量的数据类型，而不是使用变量引用对
            //// 所以下面的代码会出错，因为C#编译器发现t1是string，而t2是
            //// 即使 t2引用的是一个string,此时由于t1和t2是不同数据类型，
            //GenericMethodTest(ref t1, ref t2);
        }

        // 类型推断的Demo
        private static void GenericMethodTest<T>(ref T t1,ref T t2
        {
            T temp = t1;
            t1 = t2;
            t2 = temp;
        }
    }
}
```

代码中都有详细的注释，这里就不解释了。

二、类型约束

如果大家看了我的上一个专题的话，就应该会注意到我在实现泛型类的时候用到了where T : IComparable，在上一个专题并没有和大家介绍这个是泛型的什么用法，这个用法就是这个部分要讲的类型约束，其实where T : IComparable这句代码也很好理解的，猜猜也明白的（如果是我不知道的话，应该是猜类型参数T要满足IComparable这个接口条件，因为Where就代表符合什么条件的意思，然而真真意

思也确实如此的）下面就让我们具体看看泛型中的类型参数有哪几种约束的。 首先，编译泛型代码时，C#编译器肯定会对代码进行分析，如果我们像下面定义一个泛型类型方法时，编译器就会报错：

```
// 比较两个数的大小，返回大的那个
        private static T max<T>(T obj1, T obj2)
        {
            if (obj1.CompareTo(obj2) > 0)
            {
                return obj1;
            }

            return obj2;
        }
```

如果像上面一样定义泛型方法时，C#编译器会提示错误信息："T"不包含"CompareTo"的定义，并且找不到可接受类型为"T"的第一个参数的扩展方法"CompareTo"。 这是因为此时类型参数T可以为任意类型，然而许多类型都没有提供CompareTo方法，所以C#编译器不能编译上面的代码，这时候我们（编译器也是这么想的）肯定会想——如果C#编译器知道类型参数T有CompareTo方法的话，这样上面的代码就可以被C#编译器验证的时候通过，就不会出现编译错误的（C#编译器感觉很人性化的，都会按照人的思考方式去解决问题的，那是因为编译器也是人开发出来的，当然会人性化的，因为开发人员当时就是这么想的，所以就把逻辑写到编译器的实现中去了），这样就让我们想对类型参数作出一定约束，缩小类型参数所代表的类型数量——这就是我们类型约束的目的，从而也很自然的有了类型参数约束（这里通过对遇到的分析然后去想办法的解决的方式来引出类型约束的概念，主要是让大家可以明白**C#**中的语言特性提出来都是有原因，并不是说微软想提出来就提出来的，主要还是因为用户会有这样的需求，这样的方式我觉得可以让大家更加的明白**C#**语言特性的发展历程，从而更加深入理解**C#**，从我前面的专题也看的出来我这样介绍问题的方式的，不过这样也是我个人的理解，希望这样引入问题的方式对大家会有帮助，让大家更好的理解**C#**语言特性，如果大家对于对于有任何意见和建议的话，都可以在留言中提出的，如果觉得好的话，也麻烦表示认可下）。所以上面的代码可以指定一个类型约束，让C#编译器知道这个类型参数一定会有CompareTo方法的，这样编译器就不会报错了，我们可以将上面代码改为（代码中T:IComparable<T>为类型参数T指定的类型实参都必须实现泛型**IComparable**接口）：

```
    // 比较两个数的大小，返回大的那个
        private static T max<T>(T obj1, T obj2) **where T:IComparab
        {
            if (obj1.CompareTo(obj2) > 0)
            {
                return obj1;
            }

            return obj2;
        }
```

类型约束就是用**where** 关键字来限制能指定类型实参的类型数量，如上面的where T:IComparable<T>语句。C# 中有4种约束可以使用，然而这4种约束的语法都差不多。（约束要放在泛型方法或泛型类型声明的末尾，并且要使用Where关键字）

（1） 引用类型约束

表示形式为 **T:class**， 确保传递的类型实参必须是引用类型(注意约束的类型参数和类型本身没有关系，意思就是说定义一个泛型结构体时，泛型类型一样可以约束为引用类型，此时结构体类型本身是值类型，而类型参数约束为引用类型)，可以为任何的类、接口、委托或数组等；但是注意不能指定下面特殊的引用类型：**System.Object,System.Array,System.Delegate,System.MulticastDelegate,System.ValueType,System.Enum和System.Void.**

如下面定义的泛型类：

```
 using System.IO;
public class samplereference<T> where T : Stream
        {
             public void Test(T stream)
             {
                 stream.Close();
             }
        }
```

上面代码中类型参数T设置了引用类型约束，**Where T:stream**的意思就是告诉编译器，传入的类型实参必须是System.IO.Stream或者从Stream中派生的一个类型，如果一个类型参数没有指定约束，则默认T为**System.Object**类型（相当于一个默认约束一样，就想每个类如果没有指定构造函数就会有默认的无参数构造函数，如果指定了带参数的构造函数，编译器就不会生成一个默认的构造函数）。然而，如果我们在代码中显示指定System.Object约束时，此时会编译器会报错：约束不能是特殊类**“object**”(这里大家可以自己试试看的)

（2）值类型约束

表示形式为**T:struct**，确保传递的类型实参时值类型，其中包括枚举，但是可空类型排除，（可空类型将会在后面专题有所介绍），如下面的示例：

```
   // 值类型约束
          public class samplevaluetype<T> where T : struct
          {
              public static T Test()
              {
                  return new T();
              }
          }
```

在上面代码中，**new T()**是可以通过编译的，因为T 是一个值类型，而所有值类型都有一个公共的无参构造函数，然而，如果T不约束，或约束为引用类型时，此时上面的代码就会报错，因为有的引用类型没有公共的无参构造函数的。

（3）构造函数类型约束

表示形式为**T:new(),**如果类型参数有多个约束时，此约束必须为最后指定。确保指定的类型实参有一个公共无参构造函数的非抽象类型，这适用于：所有值类型；所有非静态、非抽象、没有显示声明的构造函数的类（前面括号中已经说了，如果显示声明带参数的构造函数，则编译器就不会为类生成一个默认的无参构造函数，大家可以通过IL反汇编程序查看下的，这里就不贴图了）；显示声明了一个公共无参构造函数的所有非抽象类。（注意： 如果同时指定构造器约束和struct约束，C#编译器会认为这是一个错误，因为这样的指定是多余的，所有值类型都隐式提供一个无参公共构造函数，就如定义接口指定访问类型为public一样，编译器也会报错，因为接口一定是public的，这样的做只多余的，所以会报错。）

（4）转换类型约束

表示形式为 **T:**基类名 （确保指定的类型实参必须是基类或派生自基类的子类）或**T:**接口名（确保指定的类型实参必须是接口或实现了该接口的类） 或**T:U**（为 T 提供的类型参数必须是为 U 提供的参数或派生自为 U 提供的参数）。转换约束的例子如下：

声明	已构造类型的例子
Class Sample<T> where T: Stream	Sample<Stream>有效的Sample<string>无效的
Class Sample<T> where T: IDisposable	Sample<Stream >有效的Sample<StringBuilder>无效的
Class Sample<T,U> where T: U	Sample<Stream,IDispsable>有效的Sample<string,IDisposable>无效的

（5）组合约束（第五种约束就是前面的4种约束的组合）

将多个不同种类的约束合并在一起的情况就是组合约束了。（注意，没有任何类型即时引用类型又是值类型的，所以引用约束和值约束不能同时使用）如果存在多个转换类型约束时，如果其中一个是类，则类必须放在接口的前面。不同的类型参数可以有不同的约束，但是他们分别要由一个单独的where关键字。下面看一些有效和无效的例子来让大家加深印象：

有效：

class Sample<T> where T:class, IDisposable, new();

class Sample<T,U> where T:class where U: struct

无效的：

class Sample<T> where T: class, struct (没有任何类型即时引用类型又是值类型的,所以为无效的)

class Sample<T> where T: Stream, class (引用类型约束应该为第一个约束，放在最前面,所以为无效的)

class Sample<T> where T: new(), Stream (构造函数约束必须放在最后面，所以为无效)

class Sample<T> where T: IDisposable, Stream(类必须放在接口前面，所以为无效的)

class Sample<T,U> where T: struct where U:class, T (类型形参“T”具有“struct”约束，因此“T”不能用作“U”的约束,所以为无效的)

class Sample<T,U> where T:Stream, U:IDisposable(不同的类型参数可以有不同的约束，但是他们分别要由一个单独的where关键字,所以为无效的)

三、利用反射调用泛型方法

下面就直接通过一个例子来演示如何利用反射来动态调用泛型方法的（关于反射的内容可以我博客中的这篇文章：http://www.cnblogs.com/zhili/archive/2012/07/08/AssemblyLoad_and_Reflection.html），演示代码如下：

```
using System;
using System.Reflection;

namespace ReflectionGenericMethod
{
    class Program
    {
        static void Main(string[] args)
        {
            Test test = new Test();
            Type type = test.GetType();

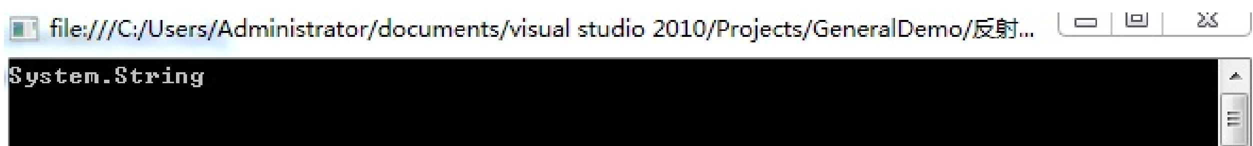
            // 首先，获得方法的定义
            // 如果不传入BindFlags实参，GetMethod方法只返回公共成员
            // 这里我指定了NonPublic，也就是返回私有成员
            // (这里要注意的是，如果指定了Public或NonPublic的话，
            // 必须要同时指定Instance|Static,否则不返回成员，具体大家可以]
            MethodInfo methodefine = type.GetMethod("PrintTypeParar
            MethodInfo constructed;

            // 使用MakeGenericMethod方法来获得一个已构造的泛型方法
            constructed = methodefine.MakeGenericMethod(typeof(str:

            // 泛型方法的调用
            constructed.Invoke(null,null);
            Console.Read();
        }
    }

    public class Test
    {
        private  static void PrintTypeParameterMethod<T>()
        {
            Console.WriteLine(typeof(T));
        }
    }
}
```

上面代码在调用泛型方法时传入的两个实参都是null,传入第一个为null是因为调用的是一个静态方法， 第二null是因为调用的方法是个无参的方法。 运行结果截图（结果是输出出 类型实参的类型，结果和我们预期的一样）：



四、小结

说到这里泛型的内容都已经介绍完了，本系列用了三个专题来介绍泛型，文章内容都基本采用提出疑问（为什么有泛型）到解释疑问，再到深入理解泛型的方式（个人认为这样的讲解方式不错的，如果大家有更好的讲解方式可以在下面留言给我），希望这种方式可以让大家知道泛型的起源，从而更好的理解泛型。后面一专题将和大家介绍了C#4.0中对泛型的改进——泛型的可变性。

泛型专题中用到的所有Demo的源代码：http://files.cnblogs.com/zhili/GeneralDemo.zip

[C# 基础知识系列]专题九: 深入理解泛型可变性

引言：

在C# 2.0中泛型并不支持可变性的（可变性指的就是协变性和逆变性),我们知道在面向对象的继承中就具有可变性，当方法声明返回类型为Stream,我们可以在实现中返回一个FileStream的类型，此时就存在一个隐式的转化——从**FileStream**类型**(**子类引用**)——>Stream**类型（父类引用），并且引用类型的数组也存在这种从子类引用**——>**父类引用的转化，例如string[] 可以转化为object[]（即这样的代码是可以通过编译的：string[] strs =new string[3]; object[] objs =strs;），此时我们肯定会想是否泛型中的泛型参数也可以支持这样的转化呢？然而在C# 2.0中是不支持的，但是就是因为有这样的需求，所以微软也考虑到这个问题的， 所以在C# 4.0中就引入了泛型的协变和逆变性。下面就具体来介绍下C# 4.0 中对协变和逆变的具体内容有哪些的。

一、协变性

协变性指的是——泛型类型参数可以从一个派生类隐式转化为基类（大家可以这样记忆的，协变性即和谐的变化，生活中我们一般会说子女长的像他们的父母，这样听起来会感觉比较和谐点，这样就很容易记住协变了），在C#4.0中引入**out**关键字来标记泛型参数支持协变性。为了更好的说明泛型的协变性，下面就以.Net类库的中**public interface IEnumerable<out T>**这个接口来演示一个例子来帮助大家理解泛型协变：

```
List<object> listobject = new List<object>();
            List<string> liststrs = new List<string>();
            // AddRange方法接收的参数类型为IEnumerable<T> collection
            // 下面的代码是传入的是List<string>类型的参数。
            // 在MSDN中可以看出这个接口的定义为——IEnumerable<int T>。
            // 所以 IEnumerable<T>泛型类型参数T支持协变性，所以可以
            // 将List<string>转化为IEnumerable<string>(这个是继承的协变
            // 又因为这个IEnumerable<in T>接口委托支持协变性，所以可以把I
            // 所以编译器验证的时候就不会出现类型不能转化的错误了。
            listobject.AddRange(liststrs);  //成功

            liststrs.AddRange(listobject); // 出错
```

代码中如果使用 这代码时 liststrs.AddRange(listobject); 就会出现编译时错误(无法从List<object>转换为IEnumerable<string>,因为List<object>可以因为继承的协变性转化为IEnumerable<object>,但是因为IEnumerable<out T>不支持逆变，即从object到string的转化，所以此时就会产生下面图中的错误了。)， 错误提示截图如下：

错误列表

2 个错误 | 0 个警告 | 0 个消息

	说明	文件	行	列	项目
1	与 “System.Collections.Generic.List<string>.AddRange(System.Collections.Generic.IEnumerable<string>)” 最匹配的重载方法具有一些无效参数	Program.cs	23	13	协变和逆变的Demo
2	参数 1: 无法从 “System.Collections.Generic.List<object>” 转换为 “System.Collections.Generic.IEnumerable<string>”	Program.cs	23	31	协变和逆变的Demo

二、逆变性

逆变性指的是——泛型类型参数可以从一个基类隐式转化为派生类(可以从生活中的例子来帮助大家记忆逆变的——如果说父母长的像他们的子女的话肯定觉得别扭,在高中语文中经常会找这样的语病的)，在C# 4.0中引入**in**关键字来标记泛型参数支持逆变性.为了更好的说明泛型的逆变性，下面就以.Net类库的中接口**public interface IComparer<in T>**来演示一个例子来帮助大家理解泛型逆变：

```
class Program
    {
        static void Main(string[] args)
        {
            List<object> listobject = new List<object>();
            List<string> liststrs = new List<string>();
            // AddRange方法接收的参数类型为IEnumerable<T> collection
            // 下面的代码是传入的是List<string>类型的参数。
            // 在MSDN中可以看出这个接口的定义为——IEnumerable<int T>。
            // 所以 IEnumerable<T>泛型类型参数T支持协变性，所以可以
            // 将List<string>转化为IEnumerable<string>(这个是继承的协变
            // 又因为这个IEnumerable<in T>接口委托支持协变性，所以可以把I
            // 所以编译器验证的时候就不会出现类型不能转化的错误了。
            listobject.AddRange(liststrs);  //成功

            ////liststrs.AddRange(listobject); // 出错

            IComparer<object> objComparer = new TestComparer();
            IComparer<string> objComparer2 = new TestComparer();

            // List<string>类型的 liststrs变量的sort方法接收的是ICompar
            // 然而下面代码传入的是 IComparer<object>这个类型的参数，要编
            // 正是因为IComparer<in T>泛型接口支持逆变，所以支持object转
            // 所以下面的这行代码可以编译通过，在.Net 4.0之前的版本肯定会编
            // 大家可以把项目的目标框架改为.Net Framework 3.5或者更加低级
            // 这样下面这行代码就会出现编译错误，因为泛型的协变和逆变是C# 4.
            liststrs.Sort(objComparer);  // 正确

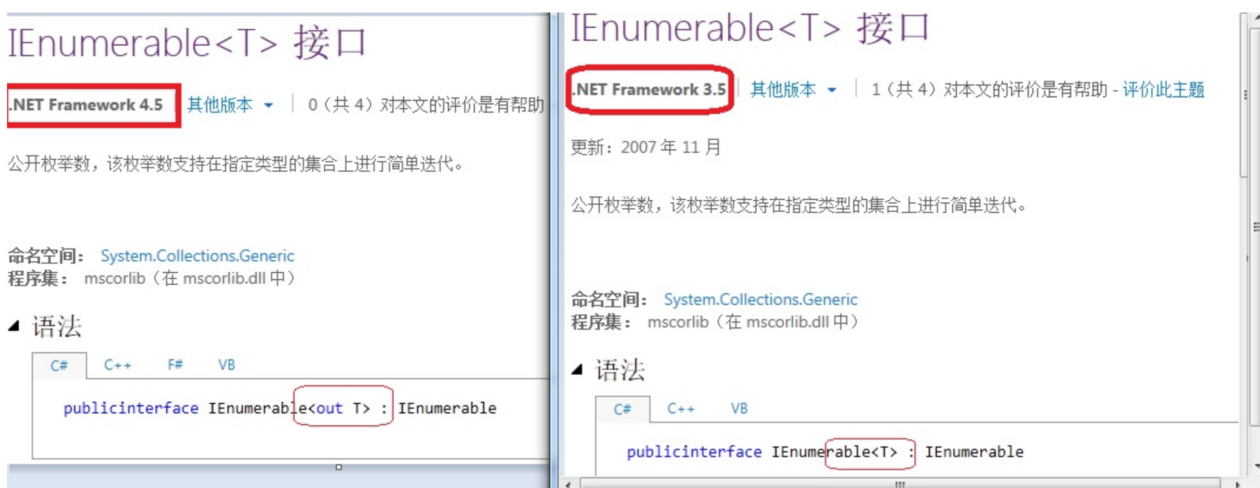
            // 出错
            ////listobject.Sort(objComparer2);
        }
    }

    public class TestComparer : IComparer<object>
    {
        public int Compare(object obj1,object obj2)
        {
            return obj1.ToString().CompareTo(obj2.ToString());
        }
    }
```

上面代码中如果使用 listobject.Sort(objComparer2);时，就会出现编译错误，错误原因看过上面协变中错误原因的解释应该都可以明白的，下面是错误的截图:



为了进一步说明泛型的协变和逆变是在C# 4.0中（C# 4.0即对于.net Framework 4.0）的版本都不支持泛型的协变和逆变,大家从MSDN中也可以发现的。下面是一张比较的截图（大家可以自己具体去MSDN上查看的， 当版本改为3.5或更低级的版本时，看下泛型的定义是不是没有out或in关键字，即之前的版本不支持泛型的可变性）：



三、协变和逆变的注意事项

并不是所有类型都支持泛型的协变和逆变的， 下面列出泛型的协变和你逆变中值得注意和明确的地方：

1. 只有接口和委托支持协变和逆变（如Func<out TResult>，Action<in T>），类或泛型方法的类型参数都不支持协变和逆变。

2. 协变和逆变只适用于引用类型，值类型不支持协变和逆变(因为可变性存在一个引用转换，而值类型变量存储的就是对象本身，而不是对象的引用)，所以List<int>无法转化为Ienumerable<object>.

3. 必须显示用in或out来标记类型参数。

4. 委托的可变性不要再多播委托中使用，相信这点很多人都没有注意到的， 下面我举个例子来说明下，当大家遇到这样的问题可以知道为什么：

上面代码可以通过编译，因为泛型Func<out T>支持协变，所以将Func<string>转换为Func<object>类型，但是对象本身仍然为Func<string>类型，然而Delegate.Combine方法要求参数必须为相同类型——否则该方法无法确定要创建什么类型的委托（是Func<string>类型呢还是Func<object>?），所以上面代码在运行时会抛出ArgumetException（错误信息为——委托必须具有相同的类型）。我们可以稍微修改下上面代码来使其不出现运行时错误

四、小结

虽然可能这个系列对实际的开发中没有多大的帮助，但是我个人认为基础还是需要打扰，只有基础打好了，才可以让我们飞的更远，更容易掌握新的技术，所以我会一直坚持下去写完这个系列的， 希望对大家巩固基础知识有所帮助。（我觉得尤其是在校学生，应该更加注重基础知识的巩固，然后写一些例子来加深对基础知识的理解）。

本专题到这里也就介绍完了（对于泛型还有一个相当有趣的话题的，就是协变和逆变的相互作用，具体这点内容大家可以参考这篇文章的：http://www.cnblogs.com/Ninputer/archive/2008/11/22/generic_covariant.html（因为我也是从这篇文章中知道这点的， 大家有兴趣的话可以去上面的链接具体看看怎么回事）），下一个专题我将和大家介绍C# 2.0中的另外一个新的特性——可空类型。

[C#基础知识系列]专题十:全面解析可空类型

引言：

C# 2.0 中还引入了可空类型，可空类型也是值类型，只是可空类型是包括null的值类型的，下面就介绍下C#2.0中对可空类型的支持具体有哪些内容(最近一直都在思考如何来分享这篇文章的,因为刚开始觉得可空类型使用过程中比较简单,觉得没有讲的必要,但是考虑到这个系列的完整性,决定还是唠叨下吧,希望对一些不熟悉的人有帮助)。

一、为什么会有可空类型

如果朋友们看了我之前的分享,对于这一部分都不会陌生,因为我一般介绍C#特性经常会以这样的方式开头的, 因为每个特性都是有它出现的原因的(有一句佛语这是这么讲的:万事皆有因,有因必有果),首先来说说这个因的(果当然是新增加了可空类型这个新特性了。)，当我们在设计数据库的时候,我们可以设置数据库字段允许为null值,如果数据库字段是日期等这样在C#语言是值类型时，当我们把数据库表映射一个对象时,此时Datetime类型在C# 语言中是不能为null的，如果这样就会与数据库的设计有所冲突，这样开发人员就会有这样的需求了——值类型能不能也为可空类型的？同时微软也看出了用户有这样的需求,所以微软在C# 2.0中就新增加了一种类型——可空类型，即包含null值的值类型，这个也就是我理解的因了，介绍完因之后，当然就是好好唠叨下可空类型是个什么东西的了？

二、可空类型的介绍

可空类型也是值类型,只是它是包含null的一个值类型。我们可以像下面这样表示可空类型（相信大家都不陌生）:

上面代码 int? 就是可空的int类型(有人可能会这样的疑问的, 如果在C#1中我硬要让一个值类型为一个可空类型怎么办到呢?当然这个在C#1之前也是有可以办到的，只是会相当麻烦，对于这个如果有兴趣的朋友可以去刨下根)，然而其实 "?"这个修饰符只是C#提供的一个语法糖(所谓语法糖,就是C#提供的一种方便的形式,其实肯定没有int? 这个类型，这个int?编译器认为的就是Nullable<int>类型，即可空类型), 其实真真C# 2.0提供的可空类型是——Nullable<T>（这个T就是上专题介绍的泛型参数,其中T只能为值类型,因为从可空类型的定义为:public struct Nullable<T> where T : struct）和Nullable。下面给出一段代码来介绍可空类型的使用：

```
namespace 可空类型Demo
{
    class Program
    {
        static void Main(string[] args)
        {
            // 下面代码也可以这样子定义int? value=1;
            Nullable<int> value = 1;

            Console.WriteLine("可空类型有值的输出情况：");
            Display(value);
            Console.WriteLine();
            Console.WriteLine();

            value = new Nullable<int>();
            Console.WriteLine("可空类型没有值的输出情况：");
            Display(value);
            Console.Read();
        }

        // 输出方法，演示可空类型中的方法和属性的使用
        private static void Display(int? nullable)
        {
            // HasValue 属性代表指示可空对象是否有值
            // 在使用Value属性时必须先判断可空类型是否有值，
            // 如果可空类型对象的HasValue返回false时，将会引发InvalidOpe
            Console.WriteLine("可空类型是否有值：{0}", nullable.HasVal
            if (nullable.HasValue)
            {
                Console.WriteLine("值为: {0}", nullable.Value);
            }

            // GetValueOrDefault(代表如果可空对象有值,就用它的值返回,如果
            // if (!nullable.HasValue)
            // {
            //    result = d.Value;
            // }

            Console.WriteLine("GetValueorDefault():{0}", nullable.G

            // GetValueOrDefault(T)方法代表如果 HasValue 属性为 true，
            Console.WriteLine("GetValueorDefalut重载方法使用：{0}", n

            // GetHashCode()代表如果 HasValue 属性为 true，则为 Value
            Console.WriteLine("GetHashCode()方法的使用：{0}", nullabl
        }
    }
}
```

输出结果:

file:///F:/学习/博客园中例子/Projects/可空类型Demo/可空类型Demo/bin/De...

```
可空类型有值的输出情况:
可空类型是否有值: True
值为: 1
GetValueorDefault():1
GetValueorDefalut重载方法使用: 1
GetHashCode()方法的使用: 1


可空类型没有值的输出情况:
可空类型是否有值: False
GetValueorDefault():0
GetValueorDefalut重载方法使用: 2
GetHashCode()方法的使用: 0
```

上面的演示代码中都注释,这里就不再解释了,为了让大家明白进一步理解可空类型是值类型,下面贴出中间语言代码截图:

```
.method private hidebysig static void  Display(valuetype [mscorlib]System.Nullable`1<int32> nullable) cil managed
{
  // 代码大小       140 (0x8c)
  .maxstack  3
  .locals init ([0] bool CS$4$0000)
  IL_0000:  nop
  IL_0001:  ldstr      bytearray (EF 53 7A 7A 7B 7C 8B 57 2F 66 26 54 09 67 3C 50   // .Szz{|.W/f&T.g<P
                                  1A FF 7B 00 30 00 7D 00 )                         // ..{.0.}.
  IL_0006:  ldarga.s   nullable
  IL_0008:  call       instance bool valuetype [mscorlib]System.Nullable`1<int32>::get_HasValue()
  IL_000d:  box        [mscorlib]System.Boolean
  IL_0012:  call       void [mscorlib]System.Console::WriteLine(string,
                                                                  object)
  IL_0017:  nop
  IL_0018:  ldarga.s   nullable
  IL_001a:  call       instance bool valuetype [mscorlib]System.Nullable`1<int32>::get_HasValue()
  IL_001f:  ldc.i4.0
```

代表可空类型为值类型

编译器把int?转化为Nullable<int32>

三、空合并操作符(**??** 操作符)

??操作符也就是"空合并操作符",它代表的意思是两个操作数,如果左边的数不为null时，就返回左边的数，如果左边的数为null,就返回右边的数，这个操作符可以用于可空类型,也可以用于引用类型,但是不能用于值类型(之所以不能应用值类型（这里除了可空类型），因为??运算符要对左边的数与null进行比较，然而值类型，不能与null类型比较，所以就不支持??运算符)，下面用一个例子来掩饰下？？运算符的使用（??这个运算符可以方便我们设置默认值，可以避免在代码中写if, else语句，简单代码数量，从而有利于阅读。）

```
static void Main(string[] args)
        {
            Console.WriteLine("??运算符的使用如下：");
            NullcoalescingOperator();
            Console.Read();
        }

        private static void NullcoalescingOperator()
        {
            int? nullable = null;
            int? nullhasvalue = 1;

            // ??和三目运算符的功能差不多的
            // 所以下面代码等价于：
            // x=nullable.HasValue?b.Value:12;
            int x = nullable ?? 12;

            // 此时nullhasvalue不能null,所以y的值为nullhasvalue.Value,
            int y = nullhasvalue ?? 123;
            Console.WriteLine("可空类型没有值的情况：{0}",x);
            Console.WriteLine("可空类型有值的情况：{0}", y);

            // 同时??运算符也可以用于引用类型，  下面是引用类型的例子
            Console.WriteLine();
            string stringnotnull = "123";
            string stringisnull = null;

            // 下面的代码等价于：
            // (stringnotnull ==null)? "456" :stringnotnull
            // 同时下面代码也等价于：
            // if(stringnotnull==null)
            // {
            //      return "456";
            // }
            // else
            // {
            //      return stringnotnull;
            // }
            // 从上面的等价代码可以看出，有了??运算符之后可以省略大量的if—e
            string result = stringnotnull ?? "456";
            string result2 = stringisnull ?? "12";
            Console.WriteLine("引用类型不为null的情况：{0}", result);
            Console.WriteLine("引用类型为null的情况：{0}", result2);
        }
```

下面是运行结果截图:

file:///F:/学习/博客园中例子/Projects/可空类型Demo/可空类型Demo/bin/De...

```
??运算符的使用如下：
可空类型没有值的情况：12
可空类型有值的情况：1

引用类型不为null的情况：123
引用类型为null的情况：12
```

四、可空类型的装箱和拆箱

值类型存在装箱和拆箱的过程,可空类型也属于值类型,从而也有装箱和拆箱的过程的, 这里先介绍下装箱和拆箱的概念的, 装箱指的的从值类型到引用类型的过程,拆箱当然也就是装箱的反过程,即从引用类型到值类型的过程(这里进一步解释下我理解的装箱和拆箱,首先.Net中值类型是分配在堆栈上的，然而引用类型分配在托管堆上,装箱过程就是把值类型的值从推栈上拷贝到托管堆上,然后推栈上存储的是对托管堆上拷贝值的引用,然而拆箱就是把托管堆上的值拷贝到堆栈上.简单一句话概况,装箱和拆箱就是一个值的拷贝的一个过程,就想搬家一样,把东西从一个地方搬到另一个地方,对于深入的理解,大家可以参考下园中的博文.), 括号中是我理解的装箱和拆箱的过程,下面就具体介绍下可空类型的装箱和拆箱的:

当把一个可空类型赋给一个引用类型变量时，此时CLR 会对可空类型（Nullable<T>）对象进行装箱处理，首先CLR会检测可空类型是否为null,如果为null,CLR则不进行实际的装箱操作（因为null可以直接赋给一个引用类型变量），如果不为null，CLR会从可空类型对象中获取值，并对该值进行装箱（这个过程就是值类型的装箱过程了。），当把一个已装箱的值类型赋给一个可空类型变量时，此时CLR会对已装箱的值类型进行拆箱处理，如果已装箱值类型的引用为null,此时CLR会把可空类型设为null（如果觉得啰嗦，大家可以直接看下面的代码，代码中也会有详细的注释）。下面用一个示例来演示下可空类型的装箱和拆箱的使用，这样可以帮助大家更好的理解前面介绍的概念:

```
static void Main(string[] args)
        {
            //Console.WriteLine("??运算符的使用如下：");
            //NullcoalescingOperator();
            Console.WriteLine("可空类型的装箱和拆箱的使用如下：");
            BoxedandUnboxed();
            Console.Read();
        }

        // 可空类型装箱和拆箱的演示
        private static void BoxedandUnboxed()
        {
            // 定义一个可空类型对象nullable
            Nullable<int> nullable = 5;
            int? nullablewithoutvalue = null;

            // 获得可空对象的类型，此时返回的是System.Int32,而不是System.
            Console.WriteLine("获取不为null的可空类型的类型为：{0}",nul

            // 对于一个为null的类型调用方法时出现异常，所以一般对于引用类型的
            //Console.WriteLine("获取为null的可空类型的类型为：{0}", nu

            // 将可空类型对象赋给引用类型obj,此时会发生装箱操作，大家可以通
            object obj = nullable;

            // 获得装箱后引用类型的类型，此时输出的仍然是System.Int32,而不
            Console.WriteLine("获得装箱后obj 的类型：{0}", obj.GetType

            // 拆箱成非可空变量
            int value = (int)obj;
            Console.WriteLine("拆箱成非可空变量的情况为：{0}", value);

            // 拆箱成可空变量
            nullable = (int?)obj;
            Console.WriteLine("拆箱成可空变量的情况为：{0}", nullable)

            // 装箱一个没有值的可空类型的对象
            obj = nullablewithoutvalue;
            Console.WriteLine("对null的可空类型装箱后obj 是否为null：{0

            // 拆箱成非可空变量,此时会抛出NullReferenceException异常,因
            // 相当于拆箱后把null值赋给一个int 类型的变量,此时当然就会出现
            //value = (int)obj;
            //Console.WriteLine("一个没有值的可空类型装箱后，拆箱成非可空

           // 拆箱成可空变量
            nullable = (int?)obj;
            Console.WriteLine("一个没有值的可空类型装箱后，拆箱成可空变量
        }
```

运行结果:

```
file:///F:/学习/博客园中例子/Projects/可空类型Demo/可空类型Demo/bin/De...
可空类型的装箱和拆箱的使用如下:
获取不为null的可空类型的类型为: System.Int32
获得装箱后obj 的类型: System.Int32
拆箱成非可空变量的情况为: 5
拆箱成可空变量的情况为: 5
对null的可空类型装箱后obj 是否为null: True
一个没有值的可空类型装箱后, 拆箱成可空变量是否为null: True
```

上面代码中都有注释的, 而且代码也比较简单, 这里就不解释了, 其实可空类型的装箱和拆箱操作大家可以就理解为非可空值类型的装箱和拆箱的过程,只是对于非可空类型因为包含null值, 所以CLR会提前对它进行检查下它是否为空, 为null就不不任何处理, 如果不为null,就按照非可空值类型的装箱和拆箱的过程来装箱和拆箱。

五、小结

到这里本专题的介绍就完成了,本专题主要介绍了下可空类型以及可空类型相关的知识,希望这篇文章可以帮助大家对可空类型的认识可以更加全面,下一个专题将和大家介绍下匿名方法, 匿名方法也是Lambda表达式和Linq的一个铺垫, 然而它是C#2中被提出来了的, 从而可以看出Lambda和Linq在C# 3.0中被添加其实是微软早在C# 2.0的时候就计划好了的, 早就计划好了的（这也是我的推断, 然而我觉得为什么它不直接在把Lambda和Linq都放在C# 2中提出来的, 却偏偏放在C# 3.0中提出, 我理解原因有——1 觉得微软当时肯定是想一起提出的, 但是后面发现这几个新的特性提出后会对编译器做比较大的改动, 需要比较长的时间来实现, 此时又怕用户等不及了, 觉得C#很多东西都没有, 所以微软就先把做好了的部分先发布出来, 然而把Lambda和Linq放到C#3来提出。我推理觉得应该是这样的, 所以C#的所有特性都是紧密相连的。）

注意：有网友提醒了我一个需要主要的点, 所以放在这里补充下, 如果细心的朋友可能会发现,当可空类型为***null***时, 此时还是可以调用***HasValue***属性, 即此时的返回值为***false,***可能就会有这样的疑问的, 为什么对象为***null***了还可以调用属性, 此时不会出现***NullReferenceException***异常吗？其实对于这个问题我之前也觉得奇怪的,后面通过查找也知道了原因了——首先,可空类型是值类型,当可空类型为***null***时, 此时可空类型并不是***null(***引用类型中的***null),***对于可空类型***null***这个是一个有效的值类型的, 所以它调用***HasValue***不会抛出异常的（值类型时不可能为***null***的, 可空类型为***null***的, 此时***null***与引用类型是不一样的, 这点大家必须明确）。同时这个问题也使我加深了对可空类型的理解, 这里分享出来可以让大家进一步理解可空类型, 如果大家有什么意见和***C#***特性需要注意的地方欢迎大家给我留言。

[C# 基础知识系列]专题十一:匿名方法解析

引言：

感觉好久没有更新博客了的，真是对不住大家了。在这个专题中将介绍匿名方法，匿名方法看名字也能明白，当然就是没有名字的方法了(现实生活中也有很多这样的匿名过程,如匿名投票,匿名举报等等,相信微软在命名方面肯定是根据了生活中例子的)，然而匿名方法的理解却不是仅仅是这一句话(这句话指的是没有名字的方法)，它还有很多内容，下面就具体介绍下匿名方法有哪些内容

一、匿名方法

之前一直认为匿名方法是在C# 3.0中提出的，之前之所以这么认为主要是因为知道C# 3.0中提出了匿名类型，所以看到匿名方法就很理所当然的认为也是在C# 3.0中提出来，然而经过系统的学习C#特性后才发现匿名方法在C# 2.0 的时候就已经提出来了，从特性的提出发展中可以看出，微软的团队是非常有计划的，后面的特性其实在之前特性的提出就已经计划好，并且后面的特性都是之前特性演变而来，之所以有新特性的提出，主要是为了方便大家编写程序，减轻程序员的工作，让编译器去执行更加复杂的操作，使程序员可以把精力放在实现自己系统的业务逻辑方法（这也是微软的主要思想，也是大部分软件所强调的良好的用户体验），然而匿名方法也正是建立在**C#1.0**中委托的基础上的（同时C# 2.0中对委托有所增强，提出了泛型委托，以及委托参数的协变和逆变，具体的可以参考本系列的前面专题），下面就具体介绍下为什么说匿名方法是如何建立在委托基础之上的（委托是方法的包装，匿名方法也是方法，只是匿名方法是没有名字的方法而已，所以委托也可以包装匿名方法）。

首先，先介绍下匿名方法的概念，匿名方法——没有名字的方法（方法也就是数学中的函数的概念），匿名方法只是在我们编写的源代码中没有指定名字而已，其实编译器会帮匿名方法生成一个名字，然而就是因为在源代码中没有名字，所以匿名方法只能在定义的时候才能调用，在其他地方不能被调用（匿名方法把方法的定义和方法的实现内嵌在一起），下面通过一个例子来看看匿名方法的使用和如何与委托关联起来的：

```
namespace 匿名方法Demo
{
    class Program
    {
        // 定义投票委托
        delegate void VoteDelegate(string name);

        static void Main(string[] args)
        {
            // 实例化委托对象
            VoteDelegate votedelegate = new VoteDelegate(new Frien

            // 使用匿名方法的代码
            // 匿名方法内联了一个委托实例（可以对照上面的委托实例化的代码来理
            // 使用匿名方法后，我们就不需要定义一个Friend类以及单独定义一个
            // 这样就可以减少代码量，代码少了，阅读起来就容易多了，以至于不会
            //VoteDelegate votedelegate = delegate(string nickname
            //{
            //    Console.WriteLine("昵称为：{0} 来帮Learning Hard投
            //};

            // 通过调用托来回调Vote()方法
            votedelegate("SomeBody");
            Console.Read();
        }
    }

    public class Friend
    {
        // 朋友的投票方法
        public void Vote(string nickname)
        {
            Console.WriteLine("昵称为：{0} 来帮Learning Hard投票了", r
        }
    }
}
```

因为前段时间参加了51博客大赛，在投票阶段也拉了好多朋友来帮忙投票的，所以为了感谢他们，所以上面就以投票作为例子来引出匿名方法，注释的部分中已经解释了匿名方法的好处的，可以帮助我们减少书写代码量，便于阅读，然而上面地方可以使用匿名方法来代替委托呢？是不是所有使用委托的地方我们都需要用匿名方法去代替的呢？事实不是这样的，因为匿名方法是没有名字的方法，所以在其他地方就不能被调用，所以不具有复用作用，并且匿名方法自动形成"闭包"（如果对于闭包不理解的朋友可以参考这两个链接：http://baike.baidu.com/view/648413.htm 和http://zh.wikipedia.org/wiki/闭包_(计算机科学)），我理解的闭包大概是当一个函数中（外部函数）调用了另个一个函数（称内部函数）时，当内部函数使用了外部函数中的变量时，这样就可能会形成闭包。具体的概念可以参考上面的两个链接，关于闭包在后面部分也会给出相关的例子来帮助大家理解，由于匿名函数会形成闭

包，这就会延长变量的生命周期）。所以如果委托包装的方法相对简单（就像上面代码中只是单独一行输出语句），并且这个方法在其他地方使用的频率很低时，这时候就可以考虑用匿名方法来代替委托。

二、使用匿名方法来忽略委托参数

第一部分主要介绍了匿名方法的概念，使用以及介绍了我所理解的为什么会有匿名方法的提出（为了方便我们实例化委托实例，通过匿名方法可以内联委托实例，这样就避免额外定义一个实例方法，减少代码量，利于阅读），在这一部分中将介绍匿名方法的另外一个好处——忽略委托参数。下面通过一个示例代码来来帮助大家理解，代码中会有详细的注释，所以这里就不多说了，直接看代码了：

```
namespace 忽略委托参数Demo
{
    class Program
    {
        static void Main(string[] args)
        {
            // Timer类在应用程序中生成定期事件
            System.Timers.Timer timer = new System.Timers.Timer();

            // 该值指示是否引发Elapsed事件
            timer.Enabled = true;

            // 设置引发Elapsed事件的间隔
            timer.Interval = 1000;

            // Elapsed事件是达到间隔时发生，前面设置了时间间隔为1秒，
            // 所以每一秒就会触发Elapsed事件，从而回调timer_Elapsed方法，
            // timer.Elapsed += new System.Timers.ElapsedEventHandl

            // 此时timer_Elapsed方法中的参数根本就不需要，所以我们可以使用
            // 省略了参数后我们的代码就更加简洁了，看的多舒服啊
            // 在开发WinForm程序中我们经常会用不到委托的参数，此时就可以使用
            timer.Elapsed += delegate
            {
                Console.WriteLine(DateTime.Now);
            };

            Console.Read();
        }

        public static void timer_Elapsed(object sender, System.Time
        {
            Console.WriteLine(DateTime.Now);
        }
    }
}
```

运行结果为：

file:///F:/学习/博客园中例子/Projects/匿名方法Demo/忽略委托参数Demo/bi...

```
2012/12/1 19:39:04
2012/12/1 19:39:05
2012/12/1 19:39:06
2012/12/1 19:39:07
2012/12/1 19:39:08
2012/12/1 19:39:09
2012/12/1 19:39:10
2012/12/1 19:39:11
2012/12/1 19:39:12
2012/12/1 19:39:13
2012/12/1 19:39:14
```

上面代码使用了匿名方法来省略委托参数，然而对于编译器而言，它还是会调用委托的构造函数来实例化委托，所以如果匿名方法能转换为多个委托类型时，此时如果省略了委托参数，编译器就不知道把匿名方法转化为哪个具体的委托类型，所以此时就会出现编译时错误，此时就必须人为的指定参数来告诉编译器如何实例化委托，下面就以创建线程为例子来帮助大家理解匿名方法省略委托参数所带来的问题（因为线程的创建涉及了两个委托类型：public delegate void ThreadStart() 和 public delegaye void ParameterizedThreadStart(objec obj)）：

```
class Program
    {
        static void Main(string[] args)
        {
            new Thread(delegate()
                {
                    Console.WriteLine("线程一");
                });

            new Thread(delegate(object o)
            {
                Console.WriteLine("线程二");
            });

            new Thread(delegate
            {
                Console.WriteLine("线程三");
            });
            Console.Read();
        }
    }
```

此时第三个创建线程的代码会出现下面的编译错误：

错误列表

1 个错误 | 0 个警告 | 0 个消息

	说明	文件
1	在以下方法或属性之间的调用不明确:"System.Threading.Thread.Thread (System.Threading.ParameterizedThreadStart)"和"System.Threading.Thread.Thread (System.Threading.ThreadStart)"	Program.cs

三、在匿名方法中捕捉变量

前面介绍中提到使用匿名方法时会形成闭包，闭包指的就是在匿名方法中捕捉了变量，为了更好的理解闭包的概念，首先需要理解两个概念 ——外部变量和被捕捉的外部变量，下面通过一个例子来解释这个两个概念：

```
class Program
    {
        // 定义闭包委托
        delegate void ClosureDelegate();

        static void Main(string[] args)
        {
            closureMethod();
            Console.Read();
        }

        // 闭包方法
        private static void closureMethod()
        {
            // outVariable和capturedVariable对于匿名方法而言都是外部变量
            // 然而outVariable是未捕获的外部变量，子所以是未捕获，是因为匿
            string outVariable = "外部变量";

            //  而capturedVariable是被匿名方法捕获的外部变量
            string capturedVariable = "捕获变量";
            ClosureDelegate closuredelegate = delegate
            {
                // localvariable是匿名方法中局部变量
                string localvariable = "匿名方法局部变量";

                Console.WriteLine(capturedVariable+" "+localvariabl
            };

            // 调用委托
            closuredelegate();
        }
    }
```

一个变量被捕捉后，被匿名方法捕捉到的是真的变量，而不是创建委托实例时该变量的值，并且被匿名方法中捕捉到的变量会延长生命周期（意思是说对于一个被捕捉的变量，只要还有任何委托实例引用它，它就一直存在，而不会当委托实例调用结束后就被垃圾回收），下面通过一个具体的例子看看匿名方法是如何延长变量的生命周期的：

```
class Program
    {
        // 定义闭包委托
        delegate void ClosureDelegate();

        static void Main(string[] args)
        {
            ClosureDelegate test = CreateDelegateInstance();
            test();

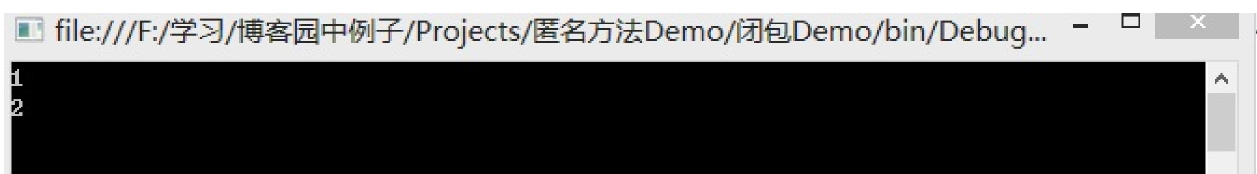
            Console.Read();
        }

        // 闭包延长变量的生命周期
        private static ClosureDelegate CreateDelegateInstance()
        {
            int count = 1;

            ClosureDelegate closuredelegate = delegate
            {
                Console.WriteLine(count);
                count++;
            };

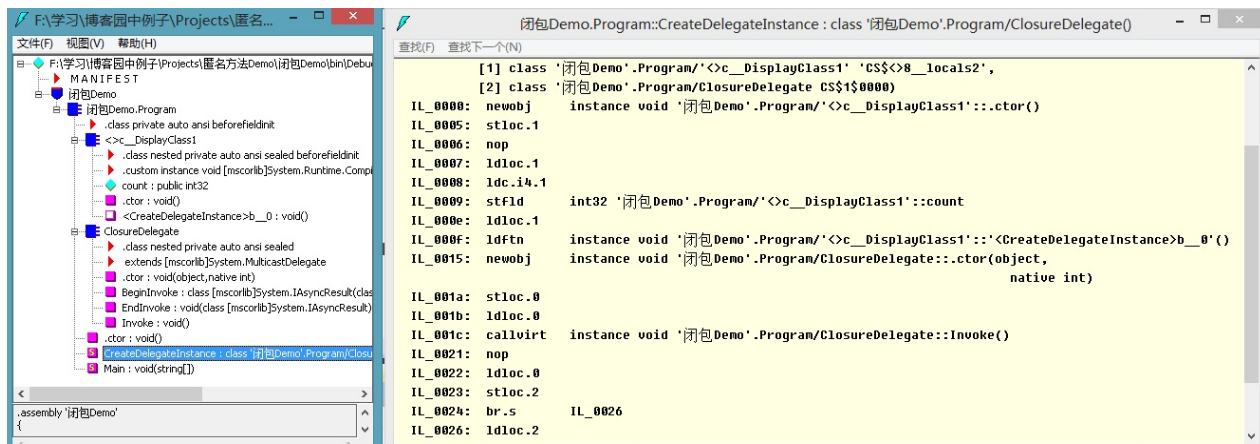
            // 调用委托
            closuredelegate();
            return closuredelegate;
        }
}
```

运行结果为：



第一行中的1是CreateDelegateInstance内部调用委托实例输出的结果，首先大家肯定认为count是在栈上分配的（因为count是值类型），当CreateDelegateInstance方法调用完后，count的值也会被销毁，当执行 test()这行代码时，此时会回调匿名方法来输出count的值，因为count被销毁，按理应该会出现异常才对的，然而结果却为2，然而结果并没有错，根据结果去倒推的话，可以得出，第二次调用委托实例也还是在使用原来的那个count，然而之所以我们认为会有异常抛出，主要原因是因为我们认为count是分配在栈上的，然而事实并不是这样的，count变量并不是分配在栈上的，事实上，编译器会创建一个额外的类来容纳变量**(此时count变量时分配在堆上的）**，**CreateDelegateInstance**方法有该类的一个实例的引用，所以此时匿名方法捕捉到的变量**count**是它的一个引用，而不是真真的值，同时匿名方法也延长了变量**count**的生命周期，使它感觉不再像是一个局部变量，反而像是一个**"全局变量"了(因为第二次中调用的委托实例使用的是同一个count)。**

匿名方法捕捉到的变量，编译器会额外创建一个类来容纳该变量，对于这点，大家可以通过**IL**反汇编程序进行查看，下面是上面程序中使用反汇编程序得到的截图：



从上面的截图中可以看出，在源代码中根本没有**<>c_DisplayClass1**类的定义的，然而这个类真是编译器为我们创建来容纳捕获变量count的，并且该类中容纳了CreateDelegateInstance方法，从上图的左半部分中间语言代码可以看出，源代码中定义的CreateDelegateInstance方法具有该**<>c_DisplayClass1**的一个引用，在源代码中使用到的count变量编译器认为是**<>c_DisplayClass1**中的一个字段。

四、小结

这个专题中主要介绍了匿名方法的使用以及匿名方法通过捕获变量来延长变量的生命周期，希望通过本专题的介绍大家可以对匿名方法可以有个全面的认识，并且匿名方法也是Lambda表达式的基础，Lambda表达式只是C# 3.0中提出更简洁的方式来实现匿名方法的。

[C#基础知识系列]专题十二:迭代器

引言:

在C# 1.0中我们经常使用foreach来遍历一个集合中的元素,然而一个类型要能够使用**foreach**关键字来对其进行遍历必须实现**IEnumerable****或IEnumerable<T>接口,(之所以来必须要实现IEnumerable这个接口,是因为**foreach**是迭代语句,要使用**foreach**必须要有一个迭代器才行的，然而**IEnumerable**接口中就有IEnumerator GetEnumerator()方法是返回迭代器的，所以实现了IEnumerable接口，就必须实现**GetEnumerator()**这个方法来返回迭代器，有了迭代器就自然就可以使用**foreach**语句了),然而在**C# 1.0**中要获得迭代器就必须实现IEnumerable接口中的**GetEnumerator()**方法，然而要实现一个迭代器就必须实现IEnumerator接口中的**bool MoveNext()**和**void Reset()**方法,然而 **C# 2.0**中提供 yield**关键字来简化迭代器的实现,这样在C# 2.0中如果我们要自定义一个迭代器就容易多了。下面就具体介绍了C# 2.0 中如何提供对迭代器的支持.

一、迭代器的介绍

迭代器大家可以想象成数据库的游标,即一个集合中的某个位置,C# 1.0中使用foreach语句实现了访问迭代器的内置支持，使用foreach使我们遍历集合更加容易（比使用for语句更加方便，并且也更加容易理解），foreach被编译后会调用GetEnumerator来返回一个迭代器，也就是一个集合中的初始位置（foreach其实也相当于是一个语法糖，把复杂的生成代码工作交给编译器去执行）。

二、**C#1.0**如何实现迭代器

在C# 1.0 中实现一个迭代器必须实现**IEnumerator**接口，下面代码演示了传统方式来实现一个自定义的迭代器：

```
 1 using System;
 2 using System.Collections;
 3 
 4 namespace 迭代器Demo
 5 {
 6     class Program
 7     {
 8         static void Main(string[] args)
 9         {
10             Friends friendcollection = new Friends();
11             foreach (Friend f in friendcollection)
12             {
13                 Console.WriteLine(f.Name);
14             }
15 
16             Console.Read();
17         }
18     }
19 
```

```
20      /// <summary>
21      ///  朋友类
22      /// </summary>
23      public class Friend
24      {
25          private string name;
26          public string Name
27          {
28              get { return name; }
29              set { name = value; }
30          }
31          public Friend(string name)
32          {
33              this.name = name;
34          }
35      }
36
37      /// <summary>
38      ///   朋友集合
39      /// </summary>
40      public class Friends : IEnumerable
41      {
42          private Friend[] friendarray;
43
44          public Friends()
45          {
46              friendarray = new Friend[]
47              {
48                  new Friend("张三"),
49                  new Friend("李四"),
50                  new Friend("王五")
51              };
52          }
53
54          // 索引器
55          public Friend this[int index]
56          {
57              get { return friendarray[index]; }
58          }
59
60          public int Count
61          {
62              get { return friendarray.Length; }
63          }
64
65          // 实现IEnumerable<T>接口方法
66         public  IEnumerator GetEnumerator()
67          {
68              return new FriendIterator(this);
69          }
70      }
71
72      /// <summary>
```

```
 73     ///  自定义迭代器，必须实现 IEnumerator接口
 74     /// </summary>
 75     public class FriendIterator : IEnumerator
 76     {
 77         private readonly Friends friends;
 78         private int index;
 79         private Friend current;
 80         internal FriendIterator(Friends friendcollection)
 81         {
 82             this.friends = friendcollection;
 83             index = 0;
 84         }
 85 
 86         #region 实现IEnumerator接口中的方法
 87         public object Current
 88         {
 89             get
 90             {
 91                 return this.current;
 92             }
 93         }
 94 
 95         public bool MoveNext()
 96         {
 97             if (index + 1 > friends.Count)
 98             {
 99                 return false;
100             }
101             else
102             {
103                 this.current = friends[index];
104                 index++;
105                 return true;
106             }
107         }
108 
109         public void Reset()
110         {
111             index = 0;
112         }
113 
114         #endregion
115     }
116 }
```

运行结果(上面代码中都有详细的注释,这里就不说明了,直接上结果截图):



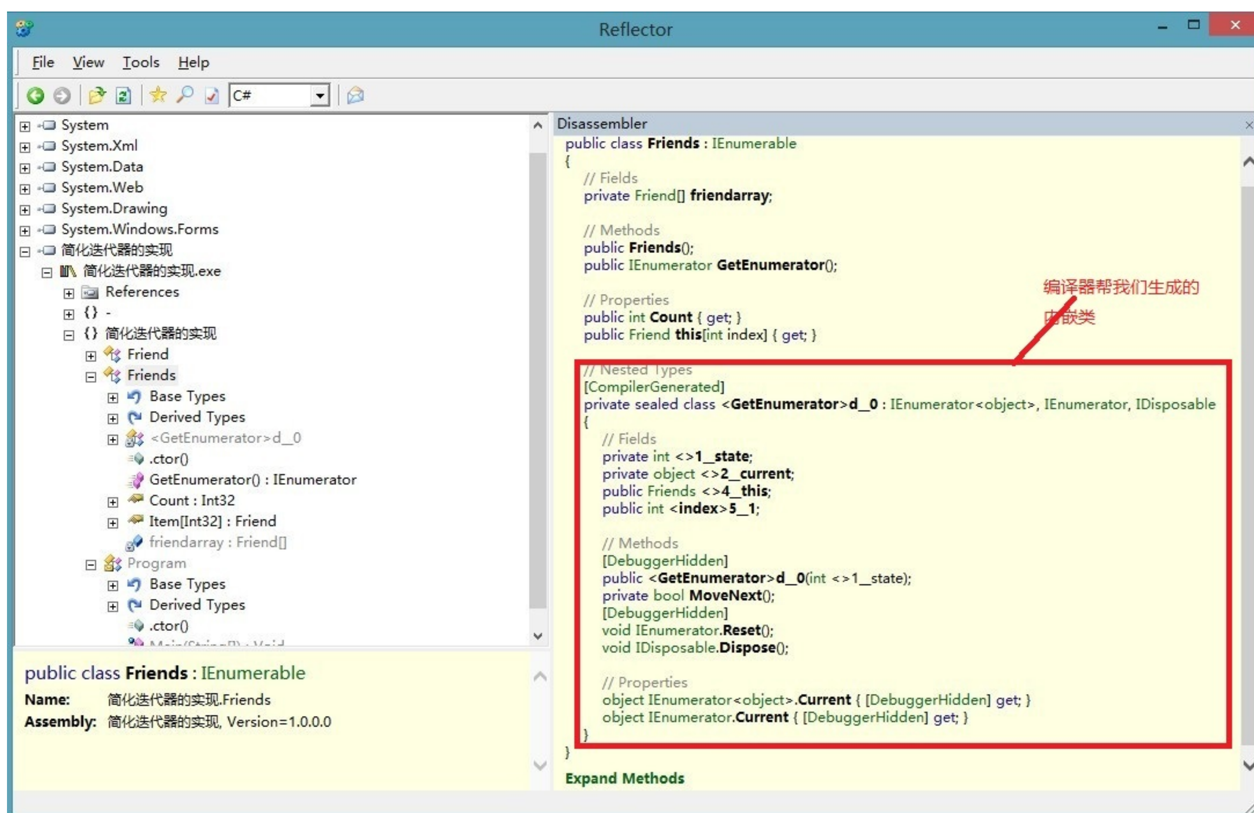
三、使用**C#2.0**的新特性简化迭代器的实现

在C# 1.0 中要实现一个迭代器必须实现**IEnumerator**接口，这样就必须实现**IEnumerator**接口中的MoveNext、Reset方法和Current属性，从上面代码中看出，为了实现FriendIterator迭代器需要写40行代码,然而在C# 2.0 中通过yield return语句简化了迭代器的实现，下面看看C# 2.0中简化迭代器的代码：

```
 1 namespace 简化迭代器的实现
 2 {
 3     class Program
 4     {
 5         static void Main(string[] args)
 6         {
 7             Friends friendcollection = new Friends();
 8             foreach (Friend f in friendcollection)
 9             {
10                 Console.WriteLine(f.Name);
11             }
12 
13             Console.Read();
14         }
15     }
16 
17     /// <summary>
18     ///  朋友类
19     /// </summary>
20     public class Friend
21     {
22         private string name;
23         public string Name
24         {
25             get { return name; }
26             set { name = value; }
27         }
28         public Friend(string name)
29         {
30             this.name = name;
31         }
32     }
33 
34     /// <summary>
35     ///   朋友集合
36     /// </summary>
37     public class Friends : IEnumerable
38     {
```

```
39          private Friend[] friendarray;
40
41          public Friends()
42          {
43              friendarray = new Friend[]
44              {
45                  new Friend("张三"),
46                  new Friend("李四"),
47                  new Friend("王五")
48              };
49          }
50
51          // 索引器
52          public Friend this[int index]
53          {
54              get { return friendarray[index]; }
55          }
56
57          public int Count
58          {
59              get { return friendarray.Length; }
60          }
61
62          // C# 2.0中简化迭代器的实现
63          public IEnumerator GetEnumerator()
64          {
65              for (int index = 0; index < friendarray.Length; inde
66              {
67                  // 这样就不需要额外定义一个FriendIterator迭代器来实现I
68                  // 在C# 2.0中只需要使用下面语句就可以实现一个迭代器
69                  yield return friendarray[index];
70              }
71          }
72      }
73  }
```

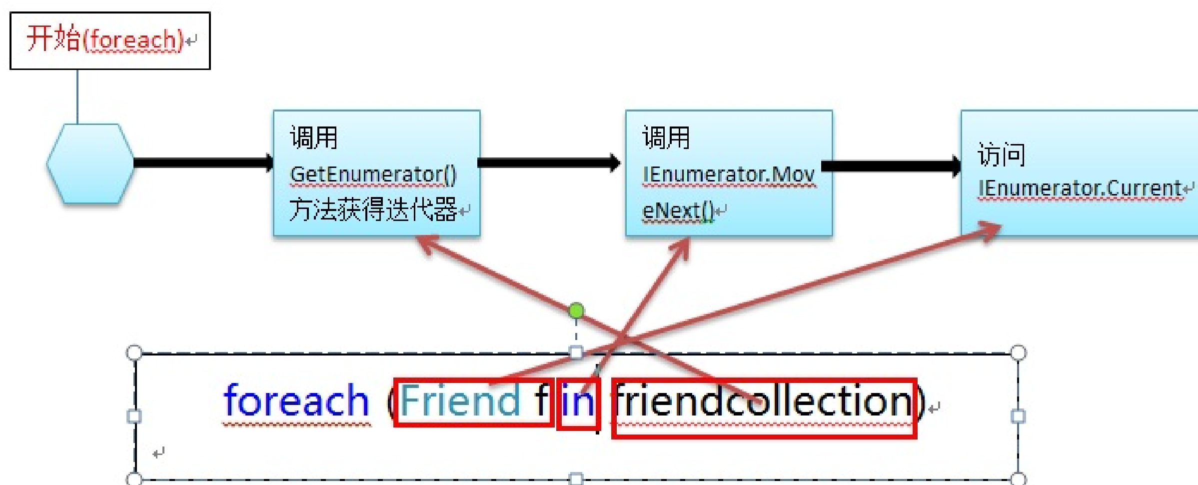
在上面代码中有一个**yield return** 语句，这个语句的作用就是告诉编译器GetEnumerator方法不是一个普通的方法，而是实现一个迭代器的方法，当编译器看到**yield return**语句时，编译器知道需要实现一个迭代器,所以编译器生成中间代码时为我们生成了一个**IEnumerator**接口的对象,大家可以通过Reflector工具进行查看,下面是通过Reflector工具得到一张截图:



从上面截图可以看出,yield return 语句其实是C#中提供的另一个语法糖，简化我们实现迭代器的源代码，把具体实现复杂迭代器的过程交给编译器帮我们去完成，看来C#编译器真是做得非常人性化，把复杂的工作留给自己做,让我们做一个简单的工作就好了。

四、迭代器的执行过程

为了让大家更好的理解迭代器,下面列出迭代器的执行流程:



五、迭代器的延迟计算

从第四部分中迭代器的执行过程中可以知道迭代器是延迟计算的, 因为迭代的主体在MoveNext()中实现（因为在MoveNext()方法中访问了集合中的当前位置的元素），Foreach中每次遍历执行到in的时候才会调用MoveNext()方法，所以迭代器

可以延迟计算,下面通过一个示例来演示迭代器的延迟计算:

```
namespace 迭代器延迟计算Demo
{
    class Program
    {
        /// <summary>
        ///  演示迭代器延迟计算
        /// </summary>
        /// <param name="args"></param>
        static void Main(string[] args)
        {
            // 测试一
            //WithIterator();
            //Console.Read();

            // 测试二
            //WithNoIterator();
            //Console.Read();

            // 测试三
            foreach (int j in WithIterator())
            {
                Console.WriteLine("在main输出语句中，当前i的值为：{0}",
            }

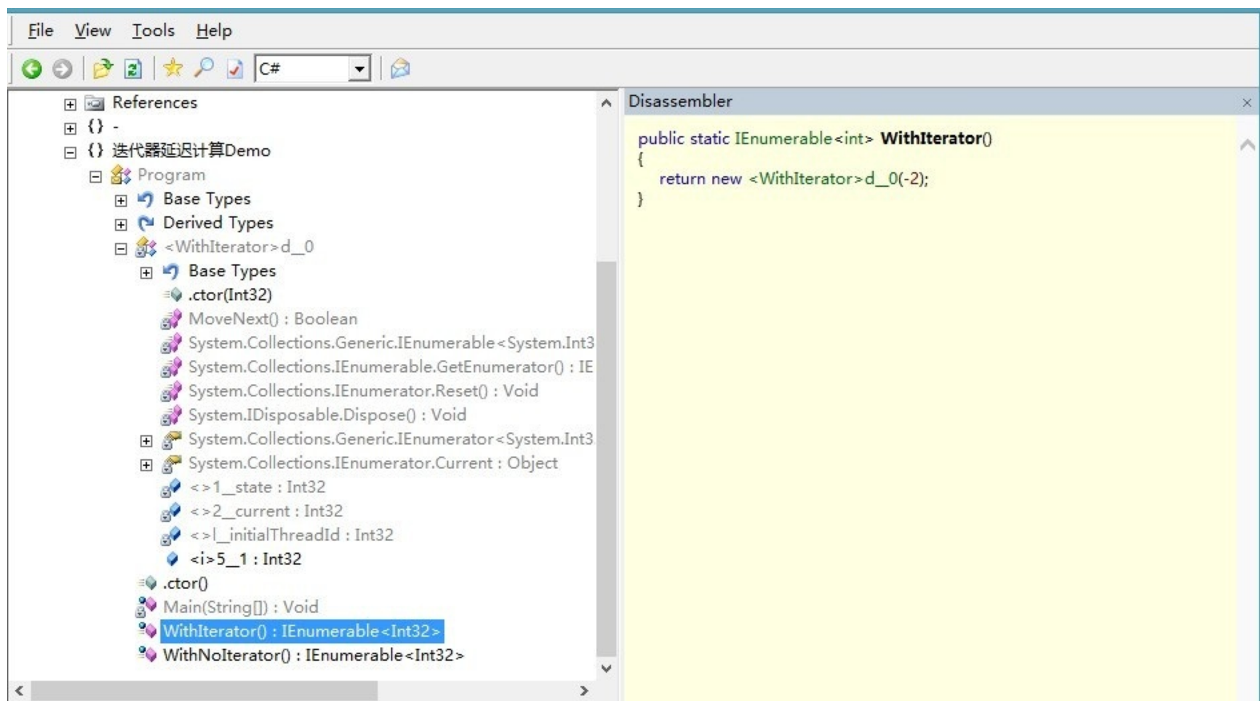
            Console.Read();
        }

        public static IEnumerable<int> WithIterator()
        {
            for (int i = 0; i < 5; i++)
            {
                Console.WriteLine("在WithIterator方法中的，当前i的值为
                if (i > 1)
                {
                    yield return i;
                }
            }
        }

        public static IEnumerable<int> WithNoIterator()
       {
           List<int> list = new List<int>();
           for (int i = 0; i < 5; i++)
           {
               Console.WriteLine("当前i的值为：{0}", i);
               if (i > 1)
               {
                   list.Add(i);
               }
           }
```

```
                return list;
            }
        }
    }
```

当运行测试一的代码时,控制台中什么都不输出,原因是生成的迭代器延迟了i 值的输出，大家可以用Reflector工具反编译出编译器生成的中间语言代码就可以发现原因了,下面是一张截图:



从图中可以看出,WithIterator()被编译成下面的代码了(此时编译器把我们自己方法体写的代码给改了):

从而当我们测试一的代码中调用WithIterator()时，对于编译器而言,就是实例化了一个**<WithIterator>d__0**的对象(<WithIterator>d__0类是编译看到WithIterator方法中包含Yield return 语句生成的一个迭代器类)，所以运行测试一的代码时，控制台中什么都不输出。

当运行测试二的代码时，运行结果就如我们期望的那样输出(这里的运行结果就不解释了，列出来是为了更好说明迭代器的延迟计算)：

```
file:///F:/学习/博客园中例子/Pr
当前i的值为: 0
当前i的值为: 1
当前i的值为: 2
当前i的值为: 3
当前i的值为: 4
```

当我们运行测试三的代码时，运行结果就有点让我们感到疑惑了， 下面先给出运行结果截图，然后在分析原因。

```
在WithIterator方法中的，当前i的值为：0
在WithIterator方法中的，当前i的值为：1
在WithIterator方法中的，当前i的值为：2
在main输出语句中，当前i的值为：2
在WithIterator方法中的，当前i的值为：3
在main输出语句中，当前i的值为：3
在WithIterator方法中的，当前i的值为：4
在main输出语句中，当前i的值为：4
```

可能刚开始看到上面的结果很多人会有疑问,为什么2,3,4会运行两次的呢?下面具体为大家分析下为什么会有这样的结果。

测试代码三中通过foreach语句来遍历集合时，当运行in的时候就会运行IEnumerator.MoveNext()方法，下面是上面代码的MoveNext()方法的代码截图：

```
private bool MoveNext()
{
    switch (this.<>1__state)
    {
        case 0:
            this.<>1__state = -1;
            this.<i>5__1 = 0;
            while (this.<i>5__1 < 10)
            {
                Console.WriteLine("当前i的值为：{0}", this.<i>5__1);
                if (this.<i>5__1 <= 5)
                {
                    goto Label_0075;
                }
                this.<>2__current = this.<i>5__1;
                this.<>1__state = 1;
                return true;
            Label_006D:
                this.<>1__state = -1;
            Label_0075:
                this.<i>5__1++;
            }
            break;

        case 1:
            goto Label_006D;
    }
    return false;
}
```

从截图中可以看到有Console.WriteLine()语句，所以用foreach遍历的时候才会有结果输出(主要是因为foreach中in 语句调用了MoveNext()方法)，至于为什么2,3,4会运行两行,主要是因为这里有两个输出语句,一个是WithIterator方法体内for语句中的输出语句,令一个是Main函数中对WithIterator方法返回的集合进行迭代的输出语句，在代码中都有明确指出，相信大家经过这样的解释后就不难理解测试三的运行结果了。

六、小结

本专题主要介绍了C# 2.0中通过**yield return**语句对迭代器实现的简化,然而对于编译器而言，却没有简化，它同样生成了一个类去实现IEnumerator接口，只是我们开发人员去实现一个迭代器得到了简化而已。希望通过本专题，大家可以对迭代器

有一个进一步的认识，并且迭代器的延迟计算也是Linq的基础，本专题之后将会和大家介绍C# 3.0中提出的新特性，然而C# 3.0中提出来的Lambda,Linq可以说是彻底改变我们编码的风格,后面的专题中将会和大家一一分享我所理解C# 3.0 中的特性。

附件：源程序代码：http://files.cnblogs.com/zhili/%E8%BF%AD%E4%BB%A3%E5%99%A8Demo.zip

破解版的Reflector工具：http://files.cnblogs.com/zhili/Reflector.zip

[C#基础知识]专题十三：全面解析对象集合初始化器、匿名类型和隐式类型

引言

经过前面专题的介绍,大家应该对C# 1和C# 2中的特性有了进一步的理解了吧,现在终于迎来我们期待已久的C# 3中特性，C# 中Lambda表达式和Linq的提出相当于彻底改变我们之前的编码风格了，刚开始接触它们，一些初学者肯定会觉得很难理解，但是我相信，只要多多研究下并且弄明白之后你肯定会爱上C# 3中的所有特性的，因为我自己就是这么过来的，在去年的这个时候，我看到Lambda表达式和Linq的时候觉得很难理解，而且觉得很奇怪的（因为之前都是用C# 3之前的特性去写代码的，虽然C# 3中的特性已经出来很久了，但是自己却写的很少，也没有怎么去研究，所以就觉得很奇怪，有一种感觉就是——怎么还可以这样写的吗？），经过这段时间对C# 语言系统的学习之后，才发现新的特性都是建立在以前特性的基础上的，只是现在编译器去帮助我们解析C# 3中提出的特性，所以对于编译器而言，用C# 3.0中的特性编写的代码和C# 2.0中编写的代码是一样的。从这个专题开始，将会为大家介绍C# 3 中的特性，本专题就介绍下C# 3中提出来的一些基础特性，这些特性也是Lambda表达式和Linq的基础。

一、自动实现的属性

当我们在类中定义的属性不需要一些额外的验证时,此时我们可以使用自动实现的属性使属性的定义更加简洁,对于C# 3中自动实现的属性,编译器编译时会创建一个私有的匿名的字段,该字段只能通过属性的get和set访问器进行访问。下面就看一个C#3中自动实现的属性的例子：

```
/// <summary>
    /// 自定义类
    /// </summary>
    public class Person
    {
        // C# 3之前我们定义属性时，一般会像下面这样去定义
        // 首先会先定义私有字段，再定义属性来对字段进行访问
        //private string _name;
        //public string Name
        //{
        //    get { return _name; }
        //    set { _name = value; }
        //}

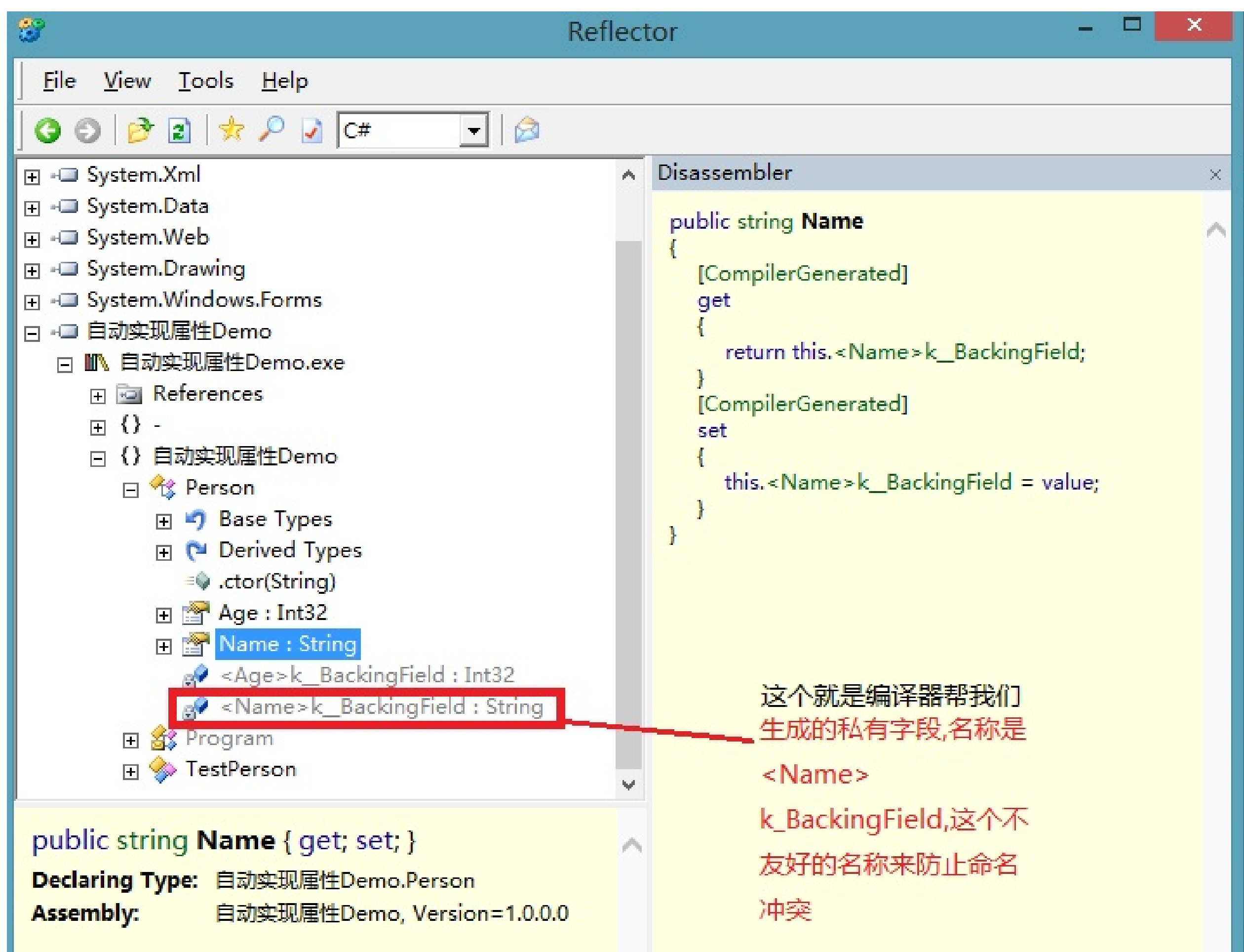
        // C# 3之后有自动实现的属性之后
        // 对于不需要额外验证的属性，就可以用自动实现的属性对属性的定义进行简化
        // 不再需要额外定义一个私有字段了，
        // 不定义私有字段并不是此时没有了私有字段，只是编译器帮我们生成一个匿名
        // 减少我们书写的代码
        // 下面就是用自动实现的属性来定义的一个属性，其效果等效于上面属性的定义

        /// <summary>
        ///  姓名
        /// </summary>
        public string Name { get; set; }

        /// <summary>
        /// 年龄
        /// </summary>
        public int Age { get; private set; }

        /// <summary>
        ///  自定义构造函数
        /// </summary>
        /// <param name="name"></param>
        public Person(string name)
        {
            Name = name;
        }
    }
```

有些人会问——你怎么知道编译器会帮我们生成一个匿名的私有字段的呢？对于这点当然通过反射工具来查看经过编译器编译之后的代码了，下面是用Reflector工具查看的一张截图：



如果在结构体中使用自动属性时,则所有构造函数都需要显式地调用无参构造函数this(),否则，就会出现编译时错误，因为只有显式调用无参构造函数this()，编译器才知道所有字段都被赋值了。下面是一段测试代码：

```
/// <summary>
    ///  在结构体使用自动属性
    /// </summary>
    public struct TestPerson
    {
        // 自动属性
        public string Name { get; set; }

        // 在结构中所有构造函数都需要显示地调用无参数构造函数this(),
        // 否则会出现编译错误
        // 只有调用了无参数构造函数，编译器才知道所有字段都被赋值了
        public TestPerson(string name)
            //: this()
        {
            this.Name = name;
        }
    }
```

把this()注释掉后就会出现编译时错误，如下图：

```
56  /// <summary> ...
59  public struct TestPerson
60  {
61      // 自动属性
62      public string Name { get; set; }
63
64      // 在结构中所有构造函数都需要显示地调用无参数构造函数this()，
65      // 否则会出现编译错误
66      // 只有调用了无参数构造函数，编译器才知道所有字段都被赋值了
67      public TestPerson(string name)
68          //: this()
69      {
70          this.Name = name;
```

错误列表

2 个错误 | 0 个警告 | 0 个消息

	说明	文件	行
1	在控制返回到调用程序之前，自动实现的属性"自动实现属性Demo.TestPerson.Name"的支持字段必须完全赋值。请考虑从构造函数初始值中调用默认构造函数。	Program.cs	67
2	在给"this"对象的所有字段赋值之前，无法使用该对象	Program.cs	70

二、隐式类型

用关键字var定义的变量则该变量就是为隐式类型，var 关键字告诉编译器根据变量的值类推断变量的类型。所以对于编译器而言，隐式类型同样也是显式的，同样具有一个显式的类型。

2.1 隐式类型的局部变量

用var 关键字来声明局部变量，下面一段演示代码：

为什么说用var定义的变量对于编译器来说还是具有显式类型呢？在Visual studio中，将鼠标放在var部分的时候就可以看到编译器为变量推断的类型。并且变量仍然是静态类型，只是我们在代码中没有写出类型的名称而已，这个工作交给编译器根据变量的值去推断出变量的类型，为了证明变量时静态类型，当我们把2赋给变量stringvariable时就会出现编译时错误，然而在其他动态语言中，这样的赋值是可以编译通过，所以用**var**声明的变量仍然还是静态类型，只是我们在代码中没有写出来而已。下面是证明上面两点的截图：

```
10  static void Main(string[] args)
11  {
12      // 用var声明局部变量
13      var stringvariable = "learning hard";
14
```

class System.String
表示文本，即一系列 Unicode 字符。

错误列表

1 个错误 | 0 个警告 | 0 个消息

	说明
1	无法将类型"int"隐式转换为"string"

然而使用隐式类型时有一些限制,具体限制有:

- 被声明的变量是一个局部变量,不能为字段(包括静态字段和实例字段)
- 变量在声明时必须被初始化(因为编译器要根据变量的赋值来推断变量的类型,如果没有被初始化则编译器就无法推断出变量类型了， 然而C#是静态语言则必须在定义变量时指定变量的类型，所以此时变量不知道什么类型，就会出现编译时错误)
- 变量的初始化不能初始化为一个方法组，也不能为一个匿名函数（前提是不进行强制类型转化的匿名函数）
- 变量不能初始化为null(因为null可以隐式转化为任何引用类型或可空类型，所以编译器不能推断出该变量到底应该为什么类型)
- 不能用一个正在声明的变量来初始化隐式类型 (如不能这样来声明隐式类型

)
- 不能用var来声明方法中的参数类型

同时使用隐式类型有优点也有缺点,下面的一段示例代码完全诠释了:

```
            // 隐式类型的优点
            // 对于复杂类型，减少打字量
            // 使用隐式类型，此时就不需要再赋值的左右两侧都指定Dictionary<s
            var dictionary = new Dictionary<string, string>();

            // 在foreach中使用隐式类型
            foreach (var item in dictionary)
            {
                //
            }

            // 隐式类型的缺点
            // 下面代码使用隐式类型就会使得开发人员很难知道变量的具体类型
            // 所以对于什么情况下使用隐式类型，完全取决个人情况，自己感觉是否
            var a = 2147483649;
            var b = 928888888888;
            var c = 2147483644;
            Console.WriteLine( "变量a的类型为：{0}",a.GetType());
            Console.WriteLine("变量b的类型为：{0}", b.GetType());
            Console.WriteLine("变量c的类型为：{0}", c.GetType());
            Console.Read();
```

2.2 隐式类型的数组

var不仅可以创建隐式类型的局部变量,还可以创建数组,下面是一段演示代码:

```
            // 隐式类型数组演示
            // 编译器推断为int[]类型
            var intarray = new[] { 1,2,3,4};

               // 编译器推断为string[] 类型
               var stringarray = new[] { "hello", "learning hard" };

               // 隐式类型数组出错的情况
               var errorarray = new[] { "hello", 3 };
```

使用隐式类型的数组时,编译器必须推断出使用什么类型的数组,编译器首先会构造一个包含大括号里面的所有表达式(如上面代码中的 1,2,3,4和"hello","learning hard")的编译时类型的集合,在这个集合中如果所有类型都能隐式转换为卫衣的一种类型，则该类型就成为数组的类型，否则，就会出现编译时错误，如代码中隐式类型数组出错的情况， 因为"hello"转化为string,而3却转化为int,此时编译器就不能确定数组的类型到底为什么,所以就会出现编译错误,错误信息为:"找不到隐式类型数组的最佳类型"

三、对象集合初始化

3.1 对象初始化

有了对象初始化特性之后,我们就不需要考虑定义参数不同的构造函数来应付不同情况的初始化了,就减少了在我们实体类中定义的构造函数代码,这样使代码更加简洁,下面就具体看下C# 3中的对象初始化的使用和注意事项:

```
namespace 对象集合初始化器Demo
{
    class Program
    {
        static void Main(string[] args)
        {
            #region 对象初始化演示
            // 在C# 3.0之前，我们可能会使用下面方式来初始化对象
            Person person1 = new Person();
            person1.Name = "learning hard";
            person1.Age = 25;

            Person person2 = new Person("learning hard");
            person2.Age = 25;

            // 如果类没有无参的构造函数就会出现编译时错误
            // 因为下面的语句是调用无参构造函数来对类中的字段进行初始化的
            // 大括号部分就是对象初始化程序
            Person person3 = new Person { Name = "learning hard", A

            // 下面代码和上面代码是等价的，只不过上面省略了构造函数的圆括号而
            Person person4 = new Person() { Name = "learning hard",
```

```
            Person person5 = new Person("learning hard") { Age = 25

            #endregion

         }
    }

    /// <summary>
    /// 自定义类
    /// </summary>
    public class Person
    {
        /// <summary>
        ///  姓名
        /// </summary>
        public string Name { get; set; }

        /// <summary>
        /// 年龄
        /// </summary>
        public int Age { get; set; }

        /// <summary>
        ///  定义无参的构造函数
        ///  如果类中自定义了带参数的构造函数,则编译不会生成默认的构造函数
        ///  如果没有默认的构造函数,则使用对象初始化时就会报错说没有实现无参的
        /// </summary>
        public Person()
        {
        }

        /// <summary>
        ///  自定义构造函数
        /// </summary>
        /// <param name="name"></param>
        public Person(string name)
        {
            Name = name;
        }
    }
}
```

上面代码中我用红色标注出使用对象初始化时需要注意的地方,大家也可以通过反射工具查看编译器是如何去解析对象初始化代码的。

3.2 集合初始化

C# 3中还提出了集合初始化特性来对集合初始化进行了优化,下面是一段集合初始化的使用演示代码:

```
namespace 对象集合初始化器Demo
```

```
{
    class Program
    {
        static void Main(string[] args)
        {

            #region 集合初始化演示

            // C# 3.0之前初始化集合使用的代码
            List<string> names = new List<string>();
            names.Add("learning hard1");
            names.Add("learning hard2");
            names.Add("learning hard3");

            // 有了C# 3.0中集合初始化特性之后，就可以简化代码
            // 同时下面也使用了隐式类型（使用了var关键字）
            var newnames = new List<string>
            {
                "learning hard1","learning hard2", "learning hard3'
            };
            #endregion
        }
    }

    /// <summary>
    /// 自定义类
    /// </summary>
    public class Person
    {
        /// <summary>
        ///  姓名
        /// </summary>
        public string Name { get; set; }

        /// <summary>
        /// 年龄
        /// </summary>
        public int Age { get; set; }

        /// <summary>
        ///  定义无参的构造函数
        ///  如果类中自定义了带参数的构造函数,则编译不会生成默认的构造函数
        ///  如果没有默认的构造函数,则使用对象初始化时就会报错说没有实现无参的
        /// </summary>
        public Person()
        {
        }
        /// <summary>
        ///  自定义构造函数
        /// </summary>
        /// <param name="name"></param>
        public Person(string name)
        {
```

```
            Name = name;
        }
    }
}
```

集合初始化同样是编译器自动帮我们调用List的无参构造函数,然后调用Add()方法一个一个地添加进去，对于编译器而言，C# 3中使用集合初始化的代码和C#3之前写的代码是一样.然而对于开发人员来说,有了C#3的集合初始化之后,这个过程就不需要我们自己去编码,而是交给编译器帮我们做就好了,为了证明编译器帮我们所做得事情,下面看看用反射工具来查看编译器到底是怎样帮我们来翻译集合初始化的:

```
    List<string> names = new List<string>();
     names.Add("learning hard1");
     names.Add("learning hard2");
     names.Add("learning hard3");
     List<string> <>g__initLocal3 = new List<string>();
     <>g__initLocal3.Add("learning hard1");
     <>g__initLocal3.Add("learning hard2");
     <>g__initLocal3.Add("learning hard3");
     List<string> newnames = <>g__initLocal3;
```

从上面反射出来的代码可以看出，编译器确实是一位大好人，帮我们做了那么多的事情。 可能大家会有这样的疑问——对象集合初始化只不过是一个语法糖而已，就是简单地让我们少写点代码而已啊,也没有其他什么用啊?下面部分的介绍将会解决你们的疑问。

四、匿名类型

看到匿名类型可能大家会联想到前面介绍的匿名方法,编译器对匿名类型和匿名方法都采用同样的处理方式,该方式为编译器为匿名类型生成类型名,我们在代码中不需要显式自定义一个类型,下面就看看匿名类型的使用:

```
namespace 匿名类型Demo
{
    class Program
    {
        static void Main(string[] args)
        {
            #region 匿名类型的使用Demo
            // 定义匿名类型
            // 因为这里不知道初始化的类型是什么，所以这里就必须使用隐式类型
            // 此时隐式类型就发挥出了功不可没的作用，从而说明隐式类型的提出是
            // 而匿名类型的提出又是服务于Linq，一步步都是在微软团队的计划当中
            Console.WriteLine("进入匿名类型使用演示：");
            var person1 = new { Name = "learning hard", Age = 25 };
            Console.WriteLine("{0} 年龄为: {1}", person1.Name, perso
            Console.Read();
            Console.WriteLine("按下Enter键进入匿名类型数组演示：");
            Console.WriteLine();
            #endregion

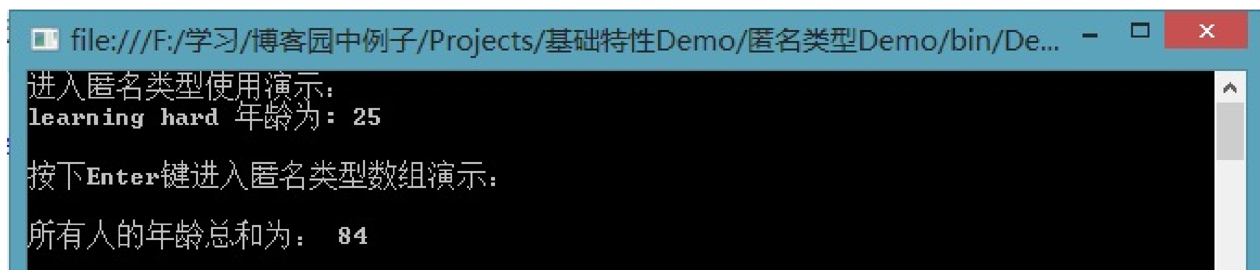
            #region 匿名类型数组演示
            // 定义匿名类型数组
            var personcollection = new[]
            {
                new {Name ="Tom",Age=30},
                new {Name ="Lily", Age=22},
                new {Name ="Jerry",Age =32},

                // 如果加入下面一句就会出现编译时错误
                // 因为此时编译器就不能推断出要转换为什么类型
                // new {Name ="learning hard"}
            };

            int totalAge = 0;
            foreach (var person in personcollection)
            {
                // 下面代码证明Age属性是强类型的int类型
                totalAge += person.Age;
            }

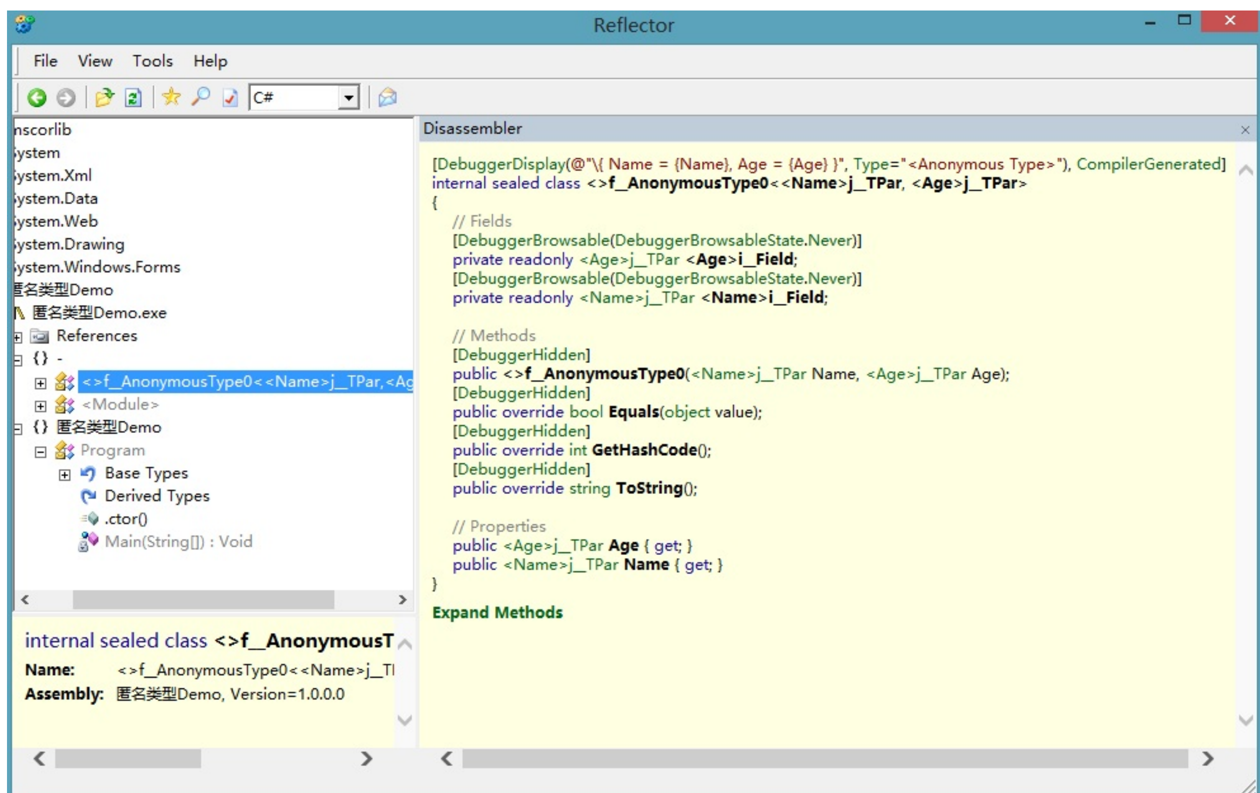
            Console.WriteLine("所有人的年龄总和为： {0}", totalAge);
            Console.ReadKey();
            #endregion
        }
    }
}
```

运行结果:



上面匿名类型的演示中使用了前面几部分介绍的所有特性——隐式类型，对象集合初始化，所以对于前面说对象集合初始化也没有其他方面的用处的疑问也可以得到答案了，如果没有对象集合初始化,要写出这样的代码(指的是 var person1 = new { Name = "learning hard", Age = 25 };)还可能吗?所以前面的隐式类型和对象集合初始化另外的一个用处就是服务于匿名类型的, 然而匿名类型又是服务于Linq的，对于Linq的好处当时是多的数不胜数了，后面专题中会为大家介绍Linq。

上面还指出虽然我们在代码中没有为匿名类型指定类型名,而编译器会为我们生成一个类型,为了证明这点我们同样反射工具Reflector查看下编译器最后为我们生成的代码到底是怎样的？截图如下:



从上面截图中可以看出编译器确实为我们生成了一个匿名类型) **<>f__AnonymousType0<<Name>j__TPar, <Age>j__TPar>**(其中代码相当于我们上面中定义的Person类),编译器为我们生成的这个类型是直接继承自System.Object的，并且是internal sealed(指的是该类型只在程序集内可见，并且不能被继承)。

五、总结

到这里，本专题的介绍也就结束了，本专题就介绍了C# 3中几个基础的特性——自动实现的属性、隐式类型、对象集合初始化和匿名类型，这些类型的提出都是服务于后面更复杂的特性Linq的，所以只有掌握好这些基础特性之后，才能更好更快地

掌握好Linq。在后面一个专题将和大家聊下C#3中的Lambda表达式。

该专题中的演示源码：http://files.cnblogs.com/zhili/%E5%9F%BA%E7%A1%80%E7%89%B9%E6%80%A7Demo.zip

[C# 基础知识系列]专题十四：深入理解Lambda表达式

引言：

对于刚刚接触Lambda表达式的朋友们，可能会对Lambda表达式感到非常疑惑，它到底是个什么什么样的技术呢?以及它有什么好处和先进的地方呢？下面的介绍将会解除你这些疑惑。

一、Lambda表达式的演变过程

Lambda表达式其实大家可以理解为它是一个匿名函数（对于匿名函数的介绍大家可以参考我这篇文章）, Lambda表达式可以包含表达式和语句,并且可以用于创建委托,以及**C#**编译器也能将它转换成表达式树。

对于Lambda表达式中都会使用这个运算符——“=>”，它读成“goes to” ,该运算符的左边为输入参数，右边是表达式或者语句块,下面就看看Lambda表达式是如何来创建委托实例（代码同时也给出了Lambda表达式从匿名方法的演示过程，从而帮助大家更好的理解Lambda表达式是匿名函数的概念，只不过C#3 中提出的Lambda表达式比匿名函数的使用更加简洁和直观了，其实原理都是一样的， 编译器同样会把Lambda表达式编译成匿名函数，也就是一个名字的方法）：

```
using System;

namespace Lambda表达式Demo
{
    class Program
    {
        /// <summary>
        /// Lambda 表达式使用演示
        /// </summary>
        /// <param name="args"></param>
        static void Main(string[] args)
        {
            // Lambda表达式的演变过程
            // 下面是C# 1中创建委托实例的代码
            Func<string, int> delegatetest1 = new Func<string, int>

            //                                    ↓
            // C# 2中用匿名方法来创建委托实例，此时就不需要额外定义回调方法C
            Func<string, int> delegatetest2 = delegate(string text)
            {
                return text.Length;
            };

            //                                    ↓
            // C# 3中使用Lambda表达式来创建委托实例
            Func<string, int> delegatetest3 = (string text) => text
```

```
            //                                        ↓
            // 可以省略参数类型string,把上面代码再简化为：
            Func<string, int> delegatetest4 = (text) => text.Lengtl

            //                                        ↓
            // 如果Lambda表达式只需一个参数，并且那个参数可以隐式指定类型时,
            // 此时可以把圆括号也省略,简化为：
            Func<string, int> delegatetest = text => text.Length;

            // 调用委托
            Console.WriteLine("使用Lambda表达式返回字符串的长度为： " +
            Console.Read();
        }

        /// <summary>
        /// 回调方法
        /// 如果使用了Lambda表达式和匿名函数，此方法就不需要额外定义
        /// </summary>
        /// <param name="text"></param>
        /// <returns></returns>
        private static int Callbackmethod(string text)
        {
            return text.Length;
        }
    }
}
```

运行结果为:



上面代码中都有详细的演变过程,这里就不多解释了,希望通过这部分之后,大家可以对Lambda表达式有进一步的理解，其实Lambda表达式就是匿名方法，其中使用Lambda表达式来创建委托实例，我们却没有指出创建的委托类型，其中编译器会帮助我们去推断委托类型，从而简化我们创建委托类型所需要的代码，从而更加简洁，所以Lambda表达式可以总结为——它是在匿名方法的基础上，再进一步地简化了创建委托实例所需要的代码。

二、Lambda表达式的使用

为了帮助大家更好的理解Lambda表达式，下面演示下用Lambda表达式来记录事件(代码中Lambda运算符的右边调用了一个回调方法ReportEvent())：

```
using System;
using System.Windows.Forms;

namespace Lambda表达式来记录事件Demo
```

```
{
    class Program
    {
        static void Main(string[] args)
        {
            // 新建一个button实例
            Button button1 = new Button() { Text ="点击我"};

            // C# 2中使用匿名方法来订阅事件
            //button1.Click+=delegate (object sender,EventArgs e)
            //{
            //    ReportEvent("Click事件", sender, e);
            //};
            //button1.KeyPress += delegate (object sender, KeyPress
            //{
            //    ReportEvent("KeyPress事件，即键盘按下事件", sender,
            //};

            // C# 3Lambda表达式方式来订阅事件
            // 与上面使用匿名方法来订阅事件是不是看出简单了很多，并且也直观了
            button1.Click += (sender, e) => ReportEvent("Click事件"
            button1.KeyPress += (sender, e) => ReportEvent("KeyPres

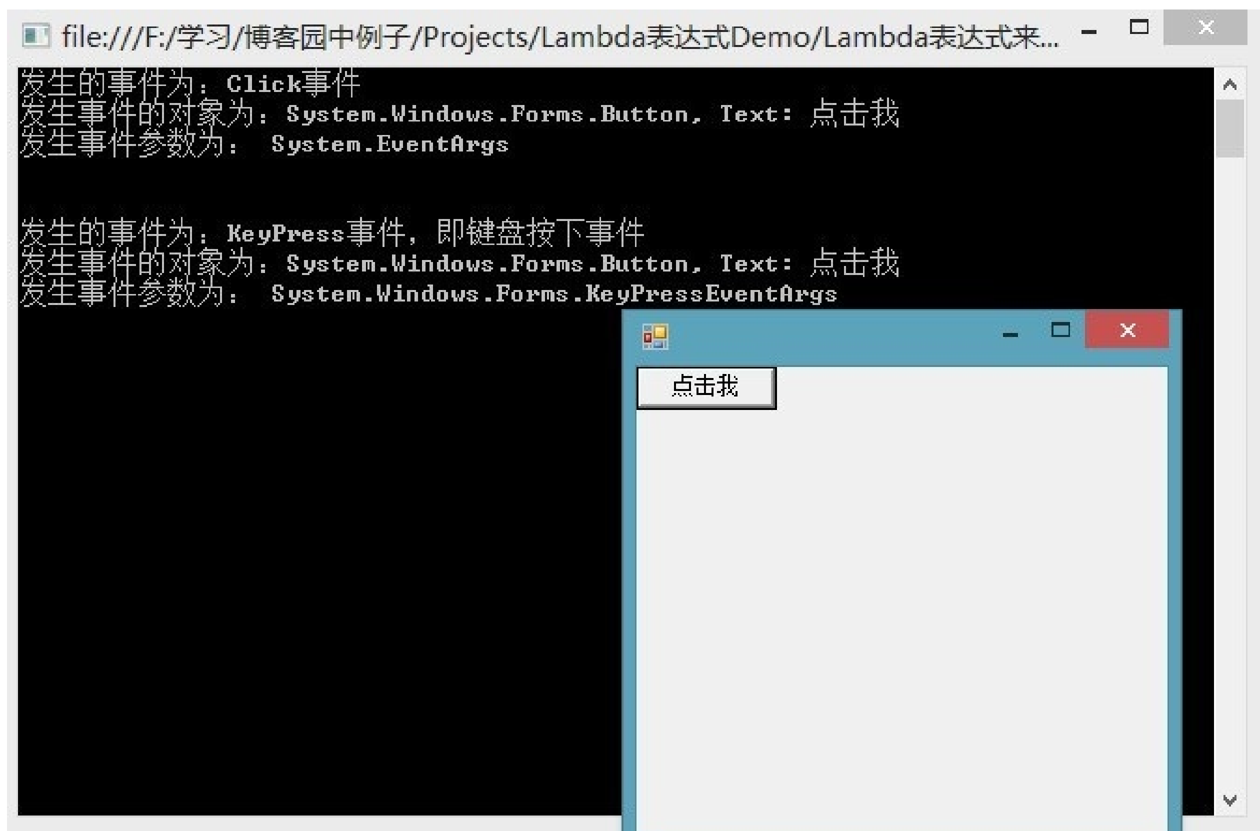
            // C# 3之前初始化对象时使用下面代码
            //Form form = new Form();
            //form.Name = "在控制台中创建的窗体";
            //form.AutoSize = true;
            //form.Controls.Add(button1);

            // C# 3中使用对象初始化器
            // 与上面代码的比较中，也可以看出使用对象初始化之后代码简化了很多
            Form form = new Form { Name = "在控制台中创建的窗体", Auto

            // 运行窗体
            Application.Run(form);
        }

        // 记录事件的回调方法
        private static void ReportEvent(string title, object sende
        {
            Console.WriteLine("发生的事件为：{0}", title);
            Console.WriteLine("发生事件的对象为：{0}", sender);
            Console.WriteLine("发生事件参数为： {0}", e.GetType());
            Console.WriteLine();
            Console.WriteLine();
        }
    }
}
```

运行结果:



从上面代码中可以看出,使用Lambda表达式之后代码确实简洁了很多，上面代码中都有详细的注释，这里就不解释了，大家可以查看代码中的注释来进行理解，并且代码中注释部分也列出了C# 3之前是如何实现这样的代码的， 这样有利于比较，从而帮助大家更好的认识到Lambda所带来的好处和进一步来理解Lambda表达式。

三、表达式树

上面指出Lambda表达式除了可以用来创建委托外，C#编译器还可以将他们转换成表达式树——用于表示Lambda表达式逻辑的一种数据结构，表达式树也可以称作表达式目录树，它将代码表示成一个对象树，而不是可执行的代码。对于刚接触哦表达式树的朋友肯定会问——为什么需要把Lambda表达式转化为表达式目录树呢？对于表达式树的提出主要是为后面Linq to SQL 做铺垫，一个Linq to SQL 的查询语句并不是在C#的程序中执行的，而是C#编译器把它转化为SQL 语句,然后再在数据库中执行。在我们使用Linq to SQL的时候都需要添加一个Linq to SQL的类，该类的扩展名dbml,该的作用就是帮助我们把Linq to SQL 的语句映射为SQL语句，然后再在数据库中执行SQL语句，把返回的结果再返回给一个IQueryable集合,所以Linq to SQL 也采用了通常的ORM（Object—Relationship—Mapping）来设计的，相当于是一个ORM框架，不过这个框架只能与微软的SQL server数据库进行映射，对于其他类型的数据库却不可以，然而很多其他开发人员却对此进行了一些扩展，扩展了对其他数据库的支持。前不久还在博客园中发布了开源的Linq框架的，名字为ELinq,其他它就是对Linq to SQL的一个扩展，使Linq语句可以映射到其他数据库的查询语句。

下面先看看如何把Lambda表达式转化为表达式目录树(其中需要引入一个新的命名空间—— **System.Linq.Expressions**)：

```
using System;
```

```
// 引用额外的命名空间
using System.Linq.Expressions;

namespace 表达式树Demo
{
    class Program
    {
        /// <summary>
        /// 表达式树的演示
        /// </summary>
        /// <param name="args"></param>
        static void Main(string[] args)
        {
            #region 将Lambda表达式转换为表达式树演示
            // 将Lambda表达式转换为Express<T>的表达式树
            // 此时express不是可执行的代码, 它现在是一个表达式树的数据结构
            Console.WriteLine("将Lambda表达式转化为表达式树的演示：");
            Expression<Func<int, int, int>> expression = (a, b) =>

            // 获得表达式树的参数
            Console.WriteLine("参数1： {0},参数2：{1}", expression.Pa

            // 既然叫做树, 那肯定有左右节点
            // 获取表达式树的主体部分
            BinaryExpression body = (BinaryExpression)expression.Bo

            // 左节点,每个节点本身就是一个表达式对象
            ParameterExpression left = (ParameterExpression)body.Le

            // 右节点
            ParameterExpression right = (ParameterExpression)body.R

            Console.WriteLine("表达式主体为：");
            Console.WriteLine(expression.Body);
            Console.WriteLine("表达式树左节点为：{0}{4} 节点类型为：{1}
            Console.Read();
            #endregion

            #region 把表达式树转化回可执行代码

            // Compile方法生成Lambda表达式的委托
            Console.WriteLine("按下Enter键进入将表达式树转换为Lambda表达
            int result = expression.Compile()(2, 3);
            Console.WriteLine("调用Lambda表达式委托结果为：" + result)
            Console.ReadKey();
            #endregion
        }
    }
}
```

运行结果:

```
file:///F:/学习/博客园中例子/Projects/Lambda表达式Demo/表达式树Demo/...
将Lambda表达式转化为表达式树的演示:
参数1:  a,参数2: b
表达式主体为:
(a + b)
表达式树左节点为: a
 节点类型为: Parameter

 表达式右节点为: b
 节点类型为: Parameter


按下Enter键进入将表达式树转换为Lambda表达式的委托演示:
调用Lambda表达式委托结果为: 5
```

上面代码首先把Lambda表达式转化为表达式树,下面这行代码就是把Lambda表达式转化为表达式树：

之后对于表达式树这种数据结构进行分析来获得该树中的主体和左右节点是什么,获得主体和左右节点的代码如下:

```
// 获取表达式树的主体部分
            BinaryExpression body = (BinaryExpression)expression.Bo

            // 左节点,每个节点本身就是一个表达式对象
            ParameterExpression left = (ParameterExpression)body.Le

            // 右节点
            ParameterExpression right = (ParameterExpression)body.R
```

从上面代码可以得出——树中的每个节点都是一个表达式(**ParameterExpression** 和**BinaryExpression**都是继承**Expression**的, 所以左右节点都是表达式),分析完表达式树之后,代码中还演示了如果把表达式树转化为可执行的代码,即转化为Lambda表达式的委托对象（此时调用**Expression**的**Compile()**方法来转化为可执行代码）, 通过调用委托来获得结果。

关于Lambda表达式树的更多信息还可以参看这篇博客：http://www.cnblogs.com/tianfan/archive/2010/03/05/expression-tree-basics.html (博主翻译的还可以)

四、总结

到这里本专题的内容也介绍的差不多了，希望通过本专题使一些之前对Lambda表达式感到疑惑的朋友们现在可以理解Lambda表达式，因为只有理解好Lambda表达式之后， 对于Linq的学习就可以说是轻而易举了。

补充：

1.匿名函数不等于匿名方法，匿名函数包含了匿名方法和lambda表达式这两种概念。 匿名函数：{匿名方法，lambda表达式} lambda作为表达式，可以被C#编译器转换为委托，也可以被编译器转换为表达式树，匿名方法只能转换为委托。 两者的共通点是都能被编译器转换成为委托，lambda表达式能完成几乎所有匿名方法能完成的事。 作为委托和表达式树，两者在IL阶段表示就不一样了。作为委托的IL，在运行期间直接被CLR所执行，而作为表达式树，是不被CLR所直接执行，而是通过相应的Provider转换为所需要的东西，比如说可以转换为SQL,也可以转换为JAVA。（引自留言中浪雪朋友的意见）

本专题中演示源码：http://files.cnblogs.com/zhili/Lambda%E8%A1%A8%E8%BE%BE%E5%BC%8FDemo.zip

[C# 基础知识系列] 专题十五：全面解析扩展方法

引言：

C# 3中所有特性的提出都是更好地为Linq服务的， 充分理解这些基础特性后。对于更深层次地去理解Linq的架构方面会更加简单，从而就可以自己去实现一个简单的ORM框架的，对于Linq的学习在下一个专题中将会简单和大家介绍下，这个专题还是先来介绍服务于Linq的基础特性——扩展方法

一、扩展方法的介绍

我一般理解一个知识点喜欢拆分去理解,所以对于扩展方法的理解可以拆分为——首先它肯定是一个方法，然而方法又是对于一个类型而言的，所以扩展方法可以理解为现有的类型(现有类型可以为自定义的类型和**.Net** 类库中的类型)扩展(添加)应该附加到该类型中的方法。

在没有扩展方法之前,如果我们想为一个已有类型自定义自己逻辑的方法时,我们必须自定义一个新的类型来继承已有类型的方式来添加方法,使用这种继承方式来添加方法时,我们必须自定义一个新的派生类型,如果基类有抽象方法还需要重新去实现抽象方法,这样为了扩展一个方法却会导致因继承而带来的其他的开销(指的是又要去自定义一个派生类，还要覆盖基类的抽象方法等)，所以使用继承来为现有类型扩展方法时就有点大才小用的感觉了，并且当我们需要为值类型和密封类（不能被继承的类）这些不能被继承的类型扩展方法时，此时继承就不能被我们所用了， 所以在C#3 中提出了用扩展方法来实现为现有类型添加方法。使用扩展方法来实现扩展可以解决使用继承中所带来的所有的弊端,下面通过一个例子来演示下扩展方法的使用：

```
class Program
    {
        /// <summary>
        ///  扩展方法演示
        /// </summary>
        /// <param name="args"></param>
        static void Main(string[] args)
        {
            #region 演示扩展方法的使用

            // 调用扩展方法
            WebRequest request = WebRequest.Create("http://www.cnbl
            using (WebResponse response = request.GetResponse())
            {
                using(Stream responsestream =response.GetResponseSt
                {
                    using (FileStream output = File.Create("respons
                    {
                        // 调用扩展方法
                        responsestream.CopyToNewStream(output);
                        Console.Read();
                    }
                }
            }
            #endregion

          }
    }

    /// <summary>
    /// 扩展方法必须在非泛型静态类中定义
    /// </summary>
    public static class StreamExten
    {
        // 定义扩展方法
        // 该扩展方法实现从一个流中内容复制到另一个流中
        public static void CopyToNewStream(this Stream inputstream,
        {
            byte[] buffer = new byte[8192];
            int read;
            while ((read = inputstream.Read(buffer, 0, buffer.Lengt
            {
                outputstream.Write(buffer, 0, read);
            }
        }
    }
```

上面程序中为Stream类型扩展了一个CopyToNewStream()的方法，然而从上面扩展方法的定义中大家可以知道扩展方法定义的一些规则，然而并不是所有方法都可以作为扩展方法来使用的, 此时朋友们就会问,我如何去分辨代码中定义的是扩展方

法还是普通的方法呢? 对于这个疑问,扩展方法的定义是要符合一些规则的,当看到定义的方法是符合这个规则,则就可以确定定义方法是扩展方法还是普通方法了。扩展方法必须具备下面的规则：

- 它必须在一个非嵌套、非泛型的静态类中
- 它至少要有一个参数
- 第一个参数必须加上this关键字作为前缀（第一个参数类型也称为扩展类型，即指方法对这个类型进行扩展）
- 第一个参数不能用其他任何修饰符（如不能使用ref out等修饰符）
- 第一个参数的类型不能是指针类型

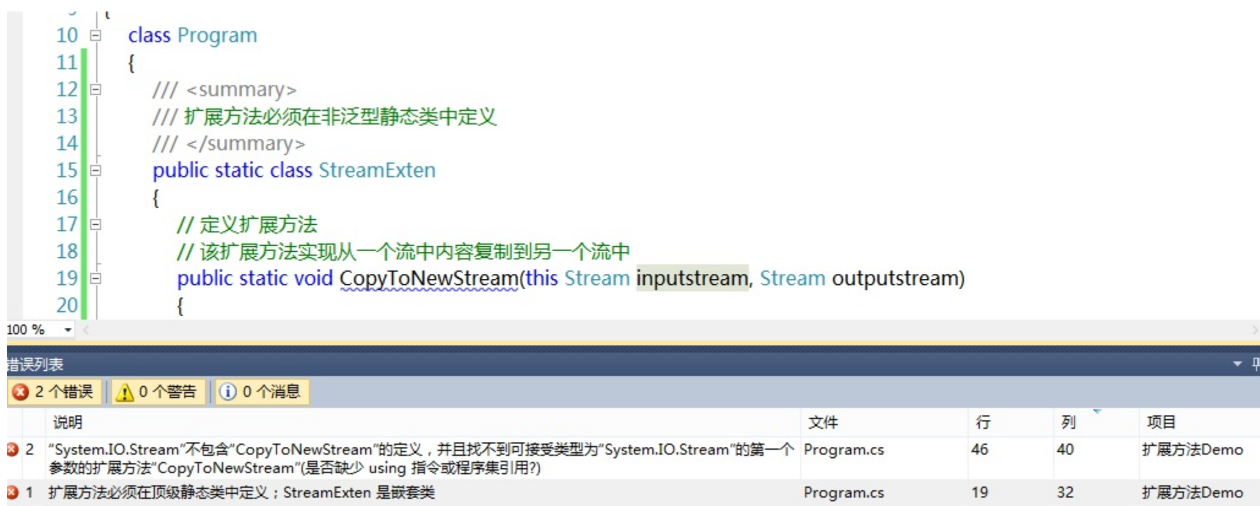
对于上面的规则大家可以在代码中试验下就会很容易明白,这些规则是一些硬性的规定,如果违反了这些规则,编译器可能会报错或者说编译器将不会认为定义的方法为扩展方法,下面简单演示下扩展方法必须在非嵌套类型的静态类中这个规则（其他规则同样大家可以在代码中进行测试），当我们把上面代码中**StreamExten** 类定义为**Program**嵌套类型时,编译器此时就会出现"扩展方法必须在顶级静态类中定义;STreamExten是嵌套类"的编译时错误，演示代码如下：

View Code

```
class Program
    {
        /// <summary>
        /// 扩展方法必须在非泛型静态类中定义
        /// </summary>
        public static class StreamExten
        {
            // 定义扩展方法
            // 该扩展方法实现从一个流中内容复制到另一个流中
            public static void CopyToNewStream(this Stream inputstr
            {
                byte[] buffer = new byte[8192];
                int read;
                while ((read = inputstream.Read(buffer, 0, buffer.L
                {
                    outputstream.Write(buffer, 0, read);
                }
            }
        }
        /// <summary>
        ///  扩展方法演示
        /// </summary>
        /// <param name="args"></param>
        static void Main(string[] args)
        {
            #region 演示扩展方法的使用

            // 调用扩展方法
            WebRequest request = WebRequest.Create("http://www.cnbl
            using (WebResponse response = request.GetResponse())
            {
                using(Stream responsestream =response.GetResponseSt
                {
                    using (FileStream output = File.Create("respons
                    {
                        // 调用扩展方法
                        responsestream.CopyToNewStream(output);
                        Console.Read();
                    }
                }
            }
            #endregion
        }
    }
```

下面是出现编译时错误截图：

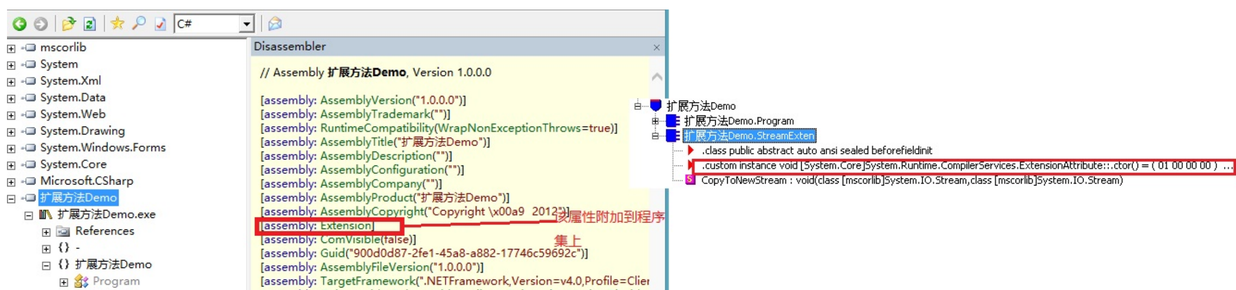


二、扩展方法是如何被发现的？

从上面部分的介绍,朋友们应该知道了如何定义和使用一个扩展方法，并且从我们定义的规则中可以帮助我们开发人员更好地去识别扩展方法，知道程序中调用的是一个实例方法还是一个扩展方法，然而相信大家此时会有这样一个疑问——编译器是如何知道我调用的是一个扩展方法而不是一个该类中的一个实例方法呢？对于这个问题，将在这部分和大家分析下。

首先讨论下程序员是如何去识别调用的是一个扩展方法而不是一个实例方法的,当我们看到调用方法的代码时,首先我们会去找该方法是否是该类(如上面程序中的Stream类)的一个实例方法，进入**Stream**类(按F12进去查看)的定义中却发现该类中没有一个名为**CopyToNewStream**的方法，此时我们就会查看程序中是否定义了这样的扩展方法，当找到一个为名CopyToNewStream这样的方法时，然后再根据定义的规则来判断找到的方法是否是为Stream类扩展的方法，这样的一个过程就是我们程序员去发现一个扩展方法的过程，然而对于编译器而言，它也是这么去发现扩展方法的（从而可以看出C#编译器还是非常智能的，完全按照人的思路去思考问题，因为它也是人实现出来的，就当然是尽可能地去以人的思考方式去实现的了），下面就介绍下编译器是如何去发现扩展方法的，这样也可以与程序员们的思路进行对比下。

当编译器看到变量调用的是一个方法时，它首先会去该对象中实例方法中去查看，一旦没有找到与调用方法同名的实例方法时，编译器就会去查找一个合适的扩展方法，它会检查导入的所有命名空间和当前的命名空间中的所有扩展方法，并匹配变量类型到扩展类型存在一个隐式转换的扩展方法。然而对于这个发现过程，可能有些人会问：编译器如何知道某个方法是扩展方法而不是实例方法呢？编译器是根据System.Runtime.CompilerServices.ExtensionAttribute属性来绑定方法是是否为扩展方法的，当我们定义的方法是扩展方法时，该属性会自动应用到方法上,编译器还会将该特性应用到包含扩展方法的程序集上,对于这个两点并不是我的推断,下面给出反编译截图来证明下:



从上面编译器发现扩展方法的过程可以得到方法调用的优先级的结论:现有的实例方法——>当前命名空间下的扩展方法——>导入命名空间的扩展方法。下面通过一个例子来演示编译器的发现过程：

```
using System;

namespace 扩展方法如何被发现Demo
{
    // 要使用不同命名空间的扩展方法首先要添加该命名空间的引用
    using CustomNamesapce;
    class Program
    {
        static void Main(string[] args)
        {
            Person p = new Person { Name = "Learning hard" };
            // 当类型中包含了实例方法时，VS中的智能提示就只会列出实例方法，而
            // 当把实例方法注释掉之后，VS的智能提示中才会列出扩展方法，此时编
            // 所以首先从当前命名空间下查找是否有该名字的扩展方法，如果找到不
            // 如果在当前命名空间中没有找到，则会到导入的命名空间中再进行查找
            p.Print();
            p.Print("Hello");
            Console.Read();
        }
    }

    // 自定义类型
    public class Person
    {
        public string Name { get; set; }

         // 当类型中的实例方法
        ////public void Print()
        ////{
        ////    Console.WriteLine("调用实例方法输出，姓名为：{0}", Name
        ////}
    }

    // 当前命名空间下的扩展方法定义
    public static class Extensionclass
    {
        /// <summary>
        ///  扩展方法定义
        /// </summary>
```

```
        /// <param name="per"></param>
        public static void Print(this Person per)
        {
            Console.WriteLine("调用的是同一命名空间下的扩展方法输出，姓名
        }
    }
}

namespace CustomNamesapce
{
    using 扩展方法如何被发现Demo;

    public static class CustomExtensionClass
    {
        /// <summary>
        ///  扩展方法定义
        /// </summary>
        /// <param name="per"></param>
        public static void Print(this Person per)
        {
            Console.WriteLine("调用的是不同命名空间下扩展方法输出，姓名为
        }

        /// <summary>
        ///  扩展方法定义
        /// </summary>
        /// <param name="per"></param>
        public static void Print(this Person per,string s)
        {
            Console.WriteLine("调用的是不同命名空间下扩展方法输出，姓名为
        }

    }
}
```

运行结果:



当没有注释掉Person类中的实例方法Print时,此时在p后面键入.运算符时，智能提示将不会出现扩展方法（扩展方法前面有一个向下的箭头标示出来的），下面是没有注释实例方法时智能提示的截图(此时智能提示不会反射扩展方法出来)：



并且从上面运行结果可以看出,当调用p.Print()方法时，此时调用的是离该调用较近的命名空间下的Print方法(尽管在CustomNamesapce命名空间下也定义了扩展方法Print)。、然而使用扩展方法还是存在一些问题的，如果同一个命名空间下的两个类都含有扩展类型相同的方法时，此时编译器就没有办法知道调用哪个方法了(这里标示出来引起大家的注意)。

三、在空引用上调用方法

大家都知道在C#中，在空引用上调用实例方法是会引发NullReferenceException异常的，但是可以在空引用上调用扩展方法，下面看一段演示代码：

```
using System;

namespace 在空引用上调用方法Demo
{
    // 必须引入扩展方法定义的命名空间
    using ExtensionDefine;

    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("空引用上调用扩展方法演示：");
            string s = null;

            // 在该程序中要使用扩展方法必须通过using来引用
            // 在空引用上调用扩展方法不会发生NullReferenceException异常
            // 之所以不会出现异常，是因为在空引用上调用扩展方法，对于编译器而
            // 对于编译器来说，s.IsNull()的调用等效于下面的代码
            //Console.WriteLine("字符串S为空字符串：{0}", NullExten.Is

            Console.WriteLine("字符串S为空字符串：{0}", s.IsNull());
            Console.ReadKey();
        }
    }
}

namespace ExtensionDefine
{
    /// <summary>
    /// 扩展方法定义
    /// </summary>
    public static class NullExten
    {
        // 此时扩展的类型为object，这里我是故意用object类型的
        // 如果是为了演示，当我们为一个类型定义扩展方法时，应尽量扩展具体类型，
        // 则所有继承于基类的类型都将具有该扩展方法，这样对其他类型来说就进行
        // 子所以形成了污染，是因为我们定义的扩展方法的意图本来只想扩展某个子
        // 其实下面这个方法我的意图只是想扩展string类型的，所以更好的定义方法
        //public static bool isNull(this string str)
        //{
        //    return str == null;
        //}

        // 不规范定义扩展方法的方式
        public static bool IsNull(this object obj)
        {
            return obj == null;
        }
    }
}
```

运行结果为:



在注释中解释了为什么在空引用中调用扩展方法不会抛出异常的原因,对于这个原因的解释也不是我个人的猜测的,而是确实如此,其实用IL反汇编程序看看程序生成的中间代码就可以证明了,下面Main函数中生成的中间代码即IL(代码中标注红色的地方就是s.IsNull()的生成的IL代码，代码意思即是调用静态类NullExten的静态方法IsNull,此时只是把空引用s传递给该方法作为传入参数，并不是真真在空引用中调用了方法。所以就不存在抛出异常了)：

```
.method private hidebysig static void  Main(string[] args) cil mana
{
  .entrypoint
  // 代码大小       43 (0x2b)
  .maxstack  2
  .locals init ([0] string s)
  IL_0000:  nop
  IL_0001:  ldstr      bytearray (7A 7A 15 5F 28 75 0A 4E 03 8C 28
                                  B9 65 D5 6C 14 6F 3A 79 1A FF )
  IL_0006:  call       void [mscorlib]System.Console::WriteLine(str
  IL_000b:  nop
  IL_000c:  ldnull
  IL_000d:  stloc.0
  IL_000e:  ldstr      bytearray (57 5B 26 7B 32 4E 53 00 3A 4E 7A
                                  32 4E 1A FF 7B 00 30 00 7D 00 )
  IL_0013:  ldloc.0
  IL_0014:  call       **bool ExtensionDefine.NullExten::IsNull(obj
**   IL_0019:  box        [mscorlib]System.Boolean
  IL_001e:  call       void [mscorlib]System.Console::WriteLine(str
                                                                obj
  IL_0023:  nop
  IL_0024:  call       valuetype [mscorlib]System.ConsoleKeyInfo [m
  IL_0029:  pop
  IL_002a:  ret
} // end of method Program::Main
```

四、小结

到这里本专题的内容就介绍完了，这里总结下该专题介绍的内容：

1. 介绍了扩展方法的定义和使用，以及扩展方法定义的规则，具体可以参照第一部分
2. 介绍了编译器是如何去发现扩展方法的，以及写了一些例子进行测试，具体可以参照第二部分
3. 解释了为什么在空引用中可以调用扩展方法的原因，具体可以参照第三部分

在下一个专题将和大家介绍下C# 3中最重要的一个特性——Linq。

附上：程序中演示源码：http://files.cnblogs.com/zhili/%E6%89%A9%E5%B1%95%E6%96%B9%E6%B3%95Demo.zip

[C# 基础知识系列]专题十六：Linq介绍

本专题概要:

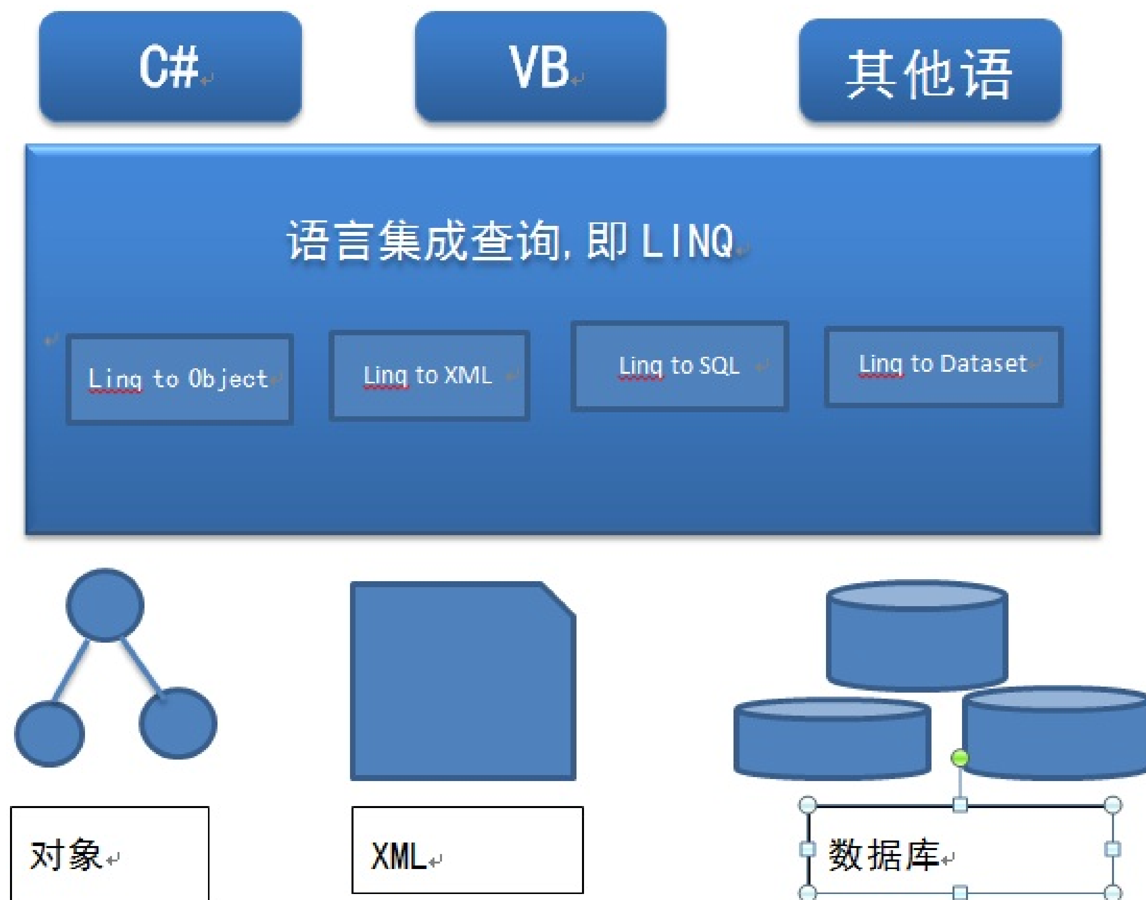
- **Linq**是什么
- 使用**Linq**的好处在哪里
- **Linq**的实际操作例子——使用**Linq**遍历文件目录
- 小结

引言：

终于到了C# 3中最重要特性的介绍了,可以说之前所有介绍的特性都是为了Linq而做准备的，然而要想深入理解Linq并不是这个专题可以介绍完的，所以我打算这个专题将对Linq做一个简单的介绍，对于Linq的深入理解我将会后面单独作为一个系列要和大家分享下。

一、**Linq**是什么？

Linq也就是Language Integrated Query的缩写,即语言集成查询,是微软在.Net 3.5中提出的一项新技术, Linq主要包含4个组件——Linq to Objects、Linq to XML、Linq to DataSet 和Linq to SQL。在这里不会具体介绍这4个组件的内容,只会给出一个大致的介绍, 下面先看看Linq的一个架构图,希望可以让大家可以对Linq有一个全面的认识:



下面简单介绍下四个组件：

- Linq to SQL 组件——可以查询基于关系数据的数据（微软本身只是实现了对SQL Server的查询，可以对数据库中的数据进行查询,修改,插入,删除,排序等操作
- Linq to Dataset组件——可以查询DasaSet对象中的数据，并对数据进行增删改查的操作
- Linq to Objects组件——可以查询IEnumberable 或IEnumberable<T>集合
- Linq to XML 组件——可以差选和操作XML文件，比Xpath操作XML更加方便

二、使用**Linq**的好处在哪里

第一部分中说到Linq中包括四个组件，分别是对不同数据进行增删改查的一些操作，然而以前也是有相关技术来对这些数据进行操作，(例如，对数据库的操作，之前有Ado.Net 对其进行支持，对XML的操作，之前也可以XPath来操作XML文件等),此时应该大家都会有个疑问的——为什么以前都有相关的技术对其进行支持，那我们为什么还需要Linq呢？对于这个疑问答案很简单，Linq 使操作这些数据源更加简单，方便和易于理解，之前的技术操作起来过于繁琐，所以微软也有上进心啊，希望可以做的更好啊，所以就在C# 3中提出了Linq来方便大家操作这些数据源，下面通过对比来说明Linq是如何简单方便：

2.1 查询集合中的数据

之前我们查询集合中的数据一般会使用for或foreach语句来进行查询，而Linq 使用查询表达式来进行查询，Linq 表达式比之前用for或forach的方式更加简洁，比较容易添加筛选条件，下面就具体看看两者方式的比较代码(我们这里假设一个场景——返回集合中序号为偶数的元素)

使用foreach 语句来返回序号为偶数的元素的实现代码如下：

```
static void Main(string[] args)
        {
            #region Linq to objects 对比
            Console.WriteLine("使用老方法来对集合对象查询，查询结果为：")
            OldQuery();
            Console.WriteLine("使用Linq方法来对集合对象查询，查询结果为：
            LinqQuery();
            Console.Read();

            #endregion
        }
       #region Linq to Objects对比

        // 使用Linq 和使用Foreach语句的对比

        // 1\. 使用foreach返回集合中序号为偶数的元素
        private static void OldQuery()
        {
            // 初始化查询的数据
            List<string> collection = new List<string>();
            for (int i = 0; i < 10; i++)
```

```
            {
                collection.Add("A"+i.ToString());
            }

            // 创建保存查询结果的集合
            List<string> queryResults = new List<string>();
            foreach (string s in collection)
            {
                // 获取元素序号
                int index = int.Parse(s.Substring(1));
                // 查询序号为偶数的元素
                if (index % 2 == 0)
                {
                    queryResults.Add(s);
                }
            }

            // 输出查询结果
            foreach (string s in queryResults)
            {
                Console.WriteLine(s);
            }
        }

        // 2\. 使用Linq返回集合中序号为偶数的元素
        private static void LinqQuery()
        {
            // 初始化查询的数据
            List<string> collection = new List<string>();
            for (int i = 0; i < 10; i++)
            {
                collection.Add("A" + i.ToString());
            }

            // 创建查询表达式来获得序号为偶数的元素
            var queryResults = from s in collection
                               let index = int.Parse(s.Substring(1)
                               where index % 2 == 0
                               select s;
            // 输出查询结果
            foreach (string s in queryResults)
            {
                Console.WriteLine(s);
            }
        }
        #endregion
```

从上面的两个方法比较中可以看出使用Linq对集合进行查询时确实简单了许多，并且也容易添加筛选条件(只需要在Where 后面添加额外的筛选条件即可)，运行结果当然也是我们期望的，下面也附上下运行结果截图：



2.2 查询XML文件

之前我们大部分都会使用XPath来对XML文件进行查询，然而使用XPath来查询XML文件需要首先知道XML文件的具体结构，而Linq 查询表达式在查询XML数据的时，可以不需要知道XML文件结构，并且编码更加简单，容易添加判断的条件，下面就具体代码来说明使用Linq查询的好处(这里假设一个场景——有一个定义Persons的XML文件，现在我们要求查找出XML文件中Name节点为“李四”的元素):

```
static void Main(string[] args)
        {
            #region Linq to XML 对比
            Console.WriteLine("使用XPath来对XML文件查询，查询结果为：");
            OldLinqToXMLQuery();
            Console.WriteLine("使用Linq方法来对XML文件查询，查询结果为：
            UsingLinqLinqtoXMLQuery();
            Console.ReadKey();
            #endregion
        }

        #region Linq to XML 对比

        // 初始化XML数据
        private static string xmlString =
            "<Persons>"+
            "<Person Id='1'>"+
            "<Name>张三</Name>"+
            "<Age>18</Age>"+
            "</Person>" +
            "<Person Id='2'>"+
            "<Name>李四</Name>"+
            "<Age>19</Age>"+
            "</Person>"+
             "<Person Id='3'>" +
            "<Name>王五</Name>" +
            "<Age>22</Age>" +
            "</Person>"+
            "</Persons>";

        // 使用XPath方式来对XML文件进行查询
        private static void OldLinqToXMLQuery()
```

```
        {
            // 导入XML文件
            XmlDocument xmlDoc = new XmlDocument();
            xmlDoc.LoadXml(xmlString);

            // 创建查询XML文件的XPath
            string xPath = "/Persons/Person";

            // 查询Person元素
            XmlNodeList querynodes = xmlDoc.SelectNodes(xPath);
            foreach (XmlNode node in querynodes)
            {
                // 查询名字为李四的元素
                foreach (XmlNode childnode in node.ChildNodes)
                {
                    if (childnode.InnerXml == "李四")
                    {
                        Console.WriteLine("姓名为： "+childnode.Inne
                    }
                }
            }

        }

        // 使用Linq 来对XML文件进行查询
        private static void UsingLinqLinqtoXMLQuery()
        {
            // 导入XML
            XElement xmlDoc = XElement.Parse(xmlString);

            // 创建查询，获取姓名为“李四”的元素
            var queryResults = from element in xmlDoc.Elements("Per
                               where element.Element("Name").Value
                               select element;

            // 输出查询结果
            foreach (var xele in queryResults)
            {
                Console.WriteLine("姓名为： " + xele.Element("Name")
            }
        }
        #endregion
```

使用XPath方式来查询XML文件时，首先需要知道XML文件的具体结构（代码中需要指定XPath为"/Persons/Person"，这就说明必须知道XML的组成结构了）,然而使用Linq方式却不需要知道XML文档结构，并且从代码书写的量上也可以看出使用Linq方式的简洁性，下面附上运行结果截图：



对于Linq to SQL 和Linq to DataSet的例子，我这里就不一一给出了，从上面的两个例子已经完全可以说明使用Linq的好处了，下面总结我理解的好处有：

- Linq 查询表达式使用上更加简单，而且也易于理解(没有接触过Linq的人也可以大致猜出代码的意图是什么的)
- Linq 提供了更多的功能，我们可以查询、排序、分组、增加和删除等操作数据的大部分功能
- 可以使用Linq处理多种数据源，也可以为特定的数据源定义自己的Linq实现（这点将会在深入理解Linq中与大家相信介绍）

三、Linq的实际操作例子——使用Linq遍历文件目录

通过前面两部分大家大致可以知道Linq的强大了吧，这部分就具体给出一个例子来看看使用Linq具体可以做些什么事情的？如果大家做一个文件管理系统的时候，大家都需要遍历文件目录的吧，下面就使用Linq来查找在文件目录中的是否存在特定的文件，具体代码如下：

```
    static void Main(string[] args)
        {
            string desktop = Environment.GetFolderPath(Environment

            //FileQuery2(desktop);
            if (!string.IsNullOrEmpty(FileQuery()))
            {
                Console.WriteLine(FileQuery());
            }
            else
            {
                Console.WriteLine("电脑桌面上不存在text.txt文件");
            }
            Console.Read();
        }
            // 使用Linq查询
        // 查询桌面是否存在text.txt文件
        private static string FileQuery()
        {
            string desktopdir = Environment.GetFolderPath(Environme

            // 获得指定目录和子目录中的文件名
            string[] filenames = Directory.GetFiles(desktopdir, "*
            List<FileInfo> files = new List<FileInfo>();
            foreach (var filename in filenames)
            {
                files.Add(new FileInfo(filename));
            }
```

```
            var results = from file in files
                          where file.Name == "text.txt"
                          select file;

            // 输出查询结果
            StringBuilder queryResult = new StringBuilder();
            foreach (var result in results)
            {
                queryResult.AppendLine("文件的路径为: " + result.Ful
            }

            return queryResult.ToString();
        }

        /// <summary>
        /// 使用递归来查找文件
        ///  查询桌面是否存在text.txt文件
        /// </summary>
        private static void FileQuery2(string path)
        {
            // 获得指定目录中的文件(包含子目录)
            string[] filenames = Directory.GetFiles(path);
            List<FileInfo> files = new List<FileInfo>();
            foreach (var filename in filenames)
            {
                files.Add(new FileInfo(filename));
            }

            var results = from file in files
                          where file.Name == "text.txt"
                          select file;

            // 输出查询结果
            StringBuilder queryResult = new StringBuilder();
            foreach (var result in results)
            {
                Console.WriteLine("文件的路径为: " + result.FullName
            }

            // 获得所有子目录
            string[] dirs = Directory.GetDirectories(path);
            if (dirs.Length > 0)
            {
                foreach (string dir in dirs)
                {
                    FileQuery2(dir);
                }
            }
        }
```

运行结果为:

```
file:///F:/学习/博客园中例子/Projects/LinqDemo/使用Linq操作文件目录Dem...
文件的路径为: C:\Users\Administrator\Desktop\text.txt
文件的路径为: C:\Users\Administrator\Desktop\mytext\text.txt
文件的路径为: C:\Users\Administrator\Desktop\mytext\text\text.txt
```

我的电脑桌面文件结果为：

四、小结

到这里本专题的内容就介绍完了，本专题主要和大家简单分享了下我对Linq的认识，希望让大家对Linq有个大概的认识，在后面的深入理解Linq系列中将会和大家一起剖析下Linq的实现原理。并且这个专题也是C# 3特性中的最后一个特性的介绍了，在后面一个专题中将带来C# 4中一个最重要的特性——动态类型(**dynamic**)的引入

专题中的源码: http://files.cnblogs.com/zhili/LinqDemo.zip

[C#基础知识系列]专题十七：深入理解动态类型

本专题概要：

- 动态类型介绍
- 为什么需要动态类型
- 动态类型的使用
- 动态类型背后的故事
- 动态类型的约束
- 实现动态行为
- 总结

引言：

终于迎来了我们C# 4中特性了，C# 4主要有两方面的改善——Com 互操作性的改进和动态类型的引入，然而COM互操作性这里就不详细介绍的，对于.Net 互操作性我将会在另外一个专题中详细和大家分享下我所了解到的知识，本专题就和大家分享C# 4中的动态类型，对于动态类型，我刚听到这个名词的时候会有这些疑问的——动态类型到底是什么的呢？ 知道动态类型大概是个什么的时候，肯定又会有这样的疑问——C# 4中为什么要引入动态类型的？(肯定引入之后可以完成我们之前不能做的事情了，肯定是有好处的)，下面就具体介绍了动态类型有哪些内容的。

一、动态类型介绍

提到动态类型当然就要说下静态类型了，对于什么是静态类型呢？ 大家都知道之前C#一直都是静态语言(指定的是没有引入动态类型之前，这里说明下，不是引入了动态类型后C#就是动态语言，只是引入动态类型后，为C#语言增添了动态语言的特性，C#仍然是静态语言)，之所以称为静态语言，之前我们写代码时，例如 int i =5;这样的代码，此时i 我们已经明确知道它的类型为int了，然而这样的代码，变量的类型的确定是在编译时确定的，对应的，如果类型的确定是在执行时才确定的类型，这样的类型就是动态类型（**C# 4.0**中新添加了一个**dynamic** 关键字来定义我们的动态类型）。面对动态类型，C#编译器做的工作只是完成检查语法是否正确，但无法确定所调用的方法或属性是否正确(之所以会这样，主要还是因为动态类型是运行时才知道它的具体类型，所以编译器编译的时候肯定不知道类型，就没办法判断调用的方法或属性是不是存在和正确了，所以对于动态类型，将不能使用VS提供的智能提示的功能，这样写动态类型代码时就要求开发人员对于某个动态类型必须准确知道其类型后和所具有的方法和属性了，不能这些错误只能在运行程序的过程抛出异常的方式被程序员所发现。)

补充： 讲到dynamic关键字，也许大家会想到C# 3中的var关键字，这里这里补充说明下dynamic, var区别。var 关键字不过是一个指令，它告诉编译器根据变量的初始化表达式来推断类型。(记住var并不是类型)，而C# 4中引入的dynamic是类型，但是编译时不属于CLR类型（指的int,string,bool,double等类型，运行时肯定CLR类型中一种的），它是包含了System.Dynamic.DynamicAttribute特性的System.Object类型，但与object又不一样，不一样主要体现在动态类型不会在编译时时执行显式转换，下面给出一段代码代码大家就会很容易看出区别了：

```
object obj = 10;
Console.WriteLine(obj.GetType());
// 使用object类型此时需要强制类型转换，不能编译器会出现编译错误
obj = (int)obj + 10;

dynamic dynamicnum = 10;
Console.WriteLine(dynamicnum.GetType());
// 对于动态类型而言，编译时编译器根本不知道它是什么类型，
// 所以编译器就判断不了dynamicnum的类型了，所以下面的代码不会出
// 因为dynamicnum有可能是int类型，编译器不知道该变量的具体类型
// 当然也就不能提示我们编译时错误了
dynamicnum = dynamicnum + 10;
```

二、为什么需要动态类型

第一部分和大家介绍了什么是动态类型，对于动态类型，总结为一句话为——运行时确定的类型。然而大家了解了动态类型到底是什么之后，当然又会出现新的问题了，即动态类型有什么用的呢？ C# 为什么好端端的引入动态类型增加程序员的负担呢？ 事实并不是这样的，下面就介绍了动态类型到底有什么用，它并不是所谓给程序员带来负担，一定程度上讲是福音

2.1 使用动态类型可以减少**强**制类型转换

从第一部分的补充也可以看到，使用动态类型不需要类型转换是因为编译器根本在编译时的过程知道什么类型，既然不知道是什么类型，怎么判断该类型是否能进行什么操作，所以也就不会出现类似"运算符"+"无法应用于"object"和"int"类型的操作数"或者"不存在int类型到某某类型的隐式转换"的编译时错误了，可能这点用户，开发人员可能并不觉得多好的，因为动态类型没有智能提示的功能。 但是动态类型减少了强制类型转换的代码之后，可读性还是会有所增强。(这里又涉及到个人取舍问题的， 如果自己觉得那种方式方便就用那种的，没必要一定要用动态类型，主要是看那种方式可以让自己和其他开发人员更好理解)

2.2 使用动态类型可以使**C#**静态语言中调用**Python**等动态语言

对于这点，可能朋友有个疑问，为什么要在C#中使用Python这样的动态语言呢？对于这个疑问，就和在C#中通过P/Invoke与本地代码交互，以及与COM互操作的道理一样，假设我们要实现的功能在C#类库中没有，然而在Python中存在时，此时我们就可以直接调用Python中存在的功能了。

三、动态类型的使用

前面两部分和大家介绍动态类型的一些基础知识的，了解完基础知识之后，大家肯定很迫不及待地想知道如何使用动态类型的，下面给出两个例子来演示动态类型的使用的。

3.1 C# 4 通过dynamic关键字来实现动态类型

```
            dynamic dyn = 5;
            Console.WriteLine(dyn.GetType());
            dyn = "test string";
            Console.WriteLine(dyn.GetType());
            dynamic startIndex = 2;
            string substring = dyn.Substring(startIndex);
            Console.WriteLine(substring);
            Console.Read();
```

运行结果为：

```
file:///F:/学习/博客园中例子/Projects/动态类型Demo/动态类型的使用Demo/...
System.Int32
System.String
st string
```

3.2 在C#中调用Python动态语言(要运行下面的代码，必须下载并安装IronPython，IronPython 是在 .NET Framework 上实现的第一种动态语言。http://ironpython.codeplex.com下载)

```
            // 引入动态类型之后
        // 可以在C#语言中与动态语言进行交互
        // 下面演示在C#中使用动态语言Python
            ScriptEngine engine = Python.CreateEngine();
            Console.Write("调用Python语言的print函数输出: ");
            // 调用Python语言的print函数来输出
            engine.Execute("print 'Hello world'");
            Console.Read();
```

运行结果：

```
file:///F:/学习/博客园中例子/Projects/动态类型Demo/动态类型的使用Demo/...
调用Python语言的print函数输出: Hello world
```

四、 动态类型背后的故事

知道了如何在C#中调用动态语言之后，然而为什么C# 为什么可以使用动态类型呢？C#编译器到底在背后为我们动态类型做了些什么事情的呢？ 对于这些问题，答案就是DLR（Dynamic Language Runtime,动态语言运行时），DLR使得C#中可以调用动态语言以及使用dynamic的动态类型。提到DLR时，可能大家会想到.Net Framework中的CLR(公共语言运行时)，然而DLR 与CLR到底是什么关系呢？下面就看看.Net 4中的组件结构图，相信大家看完之后就会明白两者之间的区别：



从图中可以看出，DLR是建立在CLR的基础之上的，其实动态语言运行时是动态语言和**C#**编译器用来动态执行代码的库，它不具有**JIT**编译，垃圾回收等功能。然而**DLR**在代码的执行过程中扮演的是什么样的角色呢？ **DLR**所扮演的角色就是——**DLR**通过它的绑定器**(binder)**和调用点**(callsite)**，元对象来把代码转换为表达式树，然后再把表达式树编译为**IL**代码，最后由**CLR**编译为本地代码（**DLR**就是帮助**C#**编译器来识别动态类型）。 这里DLR扮演的角色并不是凭空想象出来的，而且查看它的反编译代码来推出来的，下面就具体给出一个例子来说明DLR背后所做的事情。C#源代码如下：

```
class Program
    {
        static void Main(string[] args)
        {
            dynamic text = "test text";
            int startIndex = 2;
            string substring = text.Substring(startIndex);
            Console.Read();
        }
    }
```

通过Reflector工具查看生成的IL代码如下：

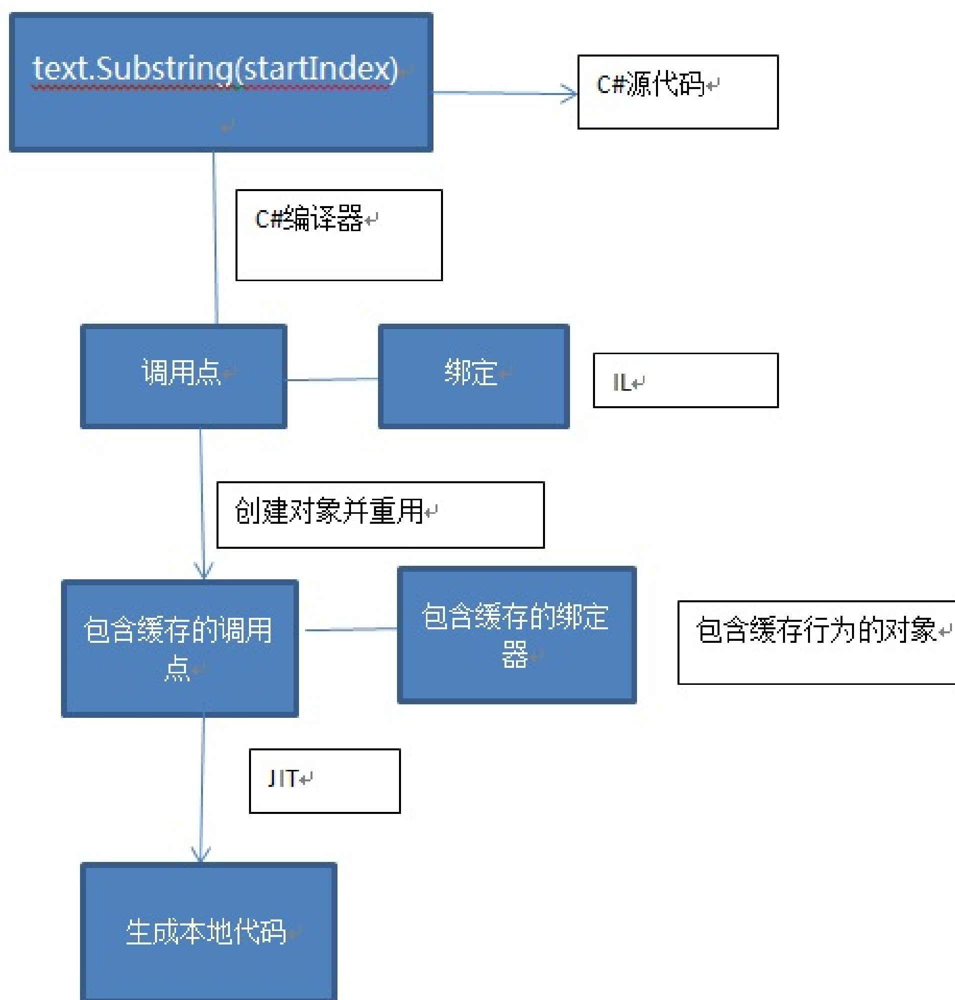
```
private static void Main(string[] args)
{
    object text = "test text";
    int startIndex = 2;
    if (<Main>o__SiteContainer0.<>p__Site1 == null)
    {
        // 创建用于将dynamic类型隐式转换为字符串的调用点
        <Main>o__SiteContainer0.<>p__Site1 = CallSite<Func<CallSite
    }
    if (<Main>o__SiteContainer0.<>p__Site2 == null)
    {
        // 创建用于调用Substring函数的调用点
        <Main>o__SiteContainer0.<>p__Site2 = CallSite<Func<CallSite
    }

    // 调用调用点，首先调用<>p_Site2,即Substring方法，再调用<>P_Site1来将
    string substring = <Main>o__SiteContainer0.<>p__Site1.Target(<M
    Console.Read();
}

//编译器生成的内嵌类型为
[CompilerGenerated]
private static class <Main>o__SiteContainer0
{
    // Fields
    public static CallSite<Func<CallSite, object, string>> <>p__Sit
    public static CallSite<Func<CallSite, object, int, object>> <>p
}
```

从IL代码中可以看出Main方法内包含两个动态操作，因为编译器生成的内嵌类型包含两个调用点(CallSite<T>,CallSite<T>即是System.Runtime.CompilerServices命名空间下的一个类，关于CallSite的具体信息可以查看MSDN中的介绍——CallSite<T>)字段，一个是调用Substring方法(即**<>p__Site2**)，一个是将结果

(编译时时dynamic)动态地转换为字符串(即<>p__Site1)，下面给出动态类型的执行过程（注意DLR中有一个缓存的概念）：



五、动态类型的约束

相信通过前面几部分的介绍大家已经对动态类型有了一定的了解的，尤其是第四部分的介绍之后，大家应该对于动态类型的执行过程也有了一个清晰的认识了，然而有些函数时不能通过动态绑定来进行调用的，这里就涉及到类型类型的约束：

5.1 不能用动态类型作为参数调用扩展方法

不能用动态类型作为参数来调用扩展方法的原因是——调用点知道编译器所知道的静态类型，但是它不知道调用所在的源文件在哪里，以及using指令引入了哪些命名空间，所以在编译时调用点就找不到哪些扩展方法可以使用，所以就会出现编译时错误。下面给出一个简单的示例程序：

```
var numbers = Enumerable.Range(10, 10);
            dynamic number = 4;
            var error = numbers.Take(number);  // 编译时错误

            // 通过下面的方式来解决这个问题
            // 1\. 将动态类型转换为正确的类型
            var right1 = numbers.Take((int)number);
            // 2\. 用调用静态方法的方式来进行调用
            var right2 = Enumerable.Take(numbers, number);
```

5.2 委托与动态类型不能隐式转换的限制

如果需要将Lambda表达式，匿名方法转化为动态类型时，此时编译器必须知道委托的确切类型，不能不加强制转化就把他们设置为Delegae或object变量，此时不同string，int类型（因为前面int,string类型可以隐式转化为动态类型，编译器此时会把他们设置为object类型。但是匿名方法和Lambda表达式不能隐式转化为动态类型），如果需要完成这样的转换，此时必须强制指定委托的类型，下面是一个演示例子：

```
dynamic lambdarestrict = x => x + 1; // 编译时错误
            // 解决方案
            dynamic rightlambda =(Func<int,int>)( x=>x+1);

            dynamic methodrestrict = Console.WriteLine; // 编译时错误
            // 解决方案
            dynamic rightmethod =(Action<string>)Console.WriteLine;
```

5.3 动态类型不能调用构造函数和静态方法的限制——即不能对动态类型调用构造函数或静态方法，因为此时编译器无法指定具体的类型。

5.4 类型声明和泛型类型参数

不能声明一个基类为dynamic的类型，也不能将dynamic用于类型参数的约束，或作为类型所实现的接口的一部分，下面看一些具体的例子来加深概念的理解：

```
 // 基类不能为dynamic 类型
    class DynamicBaseType : dynamic
    {
    }
    // dynamic类型不能为类型参数的约束
    class DynamicTypeConstrain<T> where T : dynamic
    {
    }
    // 不能作为所实现接口的一部分
    class DynamicInterface : IEnumerable<dynamic>
    {
    }
```

六、实现动态的行为

介绍了这么动态类型，是不是大家都迫不及待地想知道如果让自己的类型具有动态的行为呢？然而实现动态行为有三种方式:

- 使用ExpandObject
- 使用DynamicObject
- 实现IDynamicMetaObjectProvider接口.

下面就从最简单的方式：

6.1 使用ExpandObject来实现动态的行为

View Code

```
using System;
// 引入额外的命名空间
using System.Dynamic;

namespace 自定义动态类型
{
    class Program
    {
        static void Main(string[] args)
        {
            dynamic expand = new ExpandoObject();
            // 动态为expand类型绑定属性
            expand.Name = "Learning Hard";
            expand.Age = 24;

            // 动态为expand类型绑定方法
            expand.Addmethod = (Func<int, int>)(x => x + 1);
            // 访问expand类型的属性和方法
            Console.WriteLine("expand类型的姓名为："+expand.Name+" 年
            Console.WriteLine("调用expand类型的动态绑定的方法：" +expar
            Console.Read();
        }
    }
}
```

运行的结果和预期的一样，运行结果为：

```
file:///F:/学习/博客园中例子/Projects/动态类型Demo/自定义动态类型/bin/De...
expand类型的姓名为：Learning Hard 年龄为：  24
调用expand类型的动态绑定的方法：6
```

6.2 使用DynamicObject来实现动态行为

View Code

```
static void Main(string[] args)
       {
          dynamic dynamicobj = new DynamicType();
           dynamicobj.CallMethod();
           dynamicobj.Name = "Learning Hard";
           dynamicobj.Age = "24";
           Console.Read();
       }
 class DynamicType : DynamicObject
   {
       // 重写方法,
       // TryXXX方法表示对对象的动态调用
       public override bool TryInvokeMember(InvokeMemberBinder bir
       {
           Console.WriteLine(binder.Name +" 方法正在被调用");
           result = null;
           return true;
       }

       public override bool TrySetMember(SetMemberBinder binder, c
       {
           Console.WriteLine(binder.Name + " 属性被设置, " + "设置的
           return true;
       }
   }
```

运行结果为：



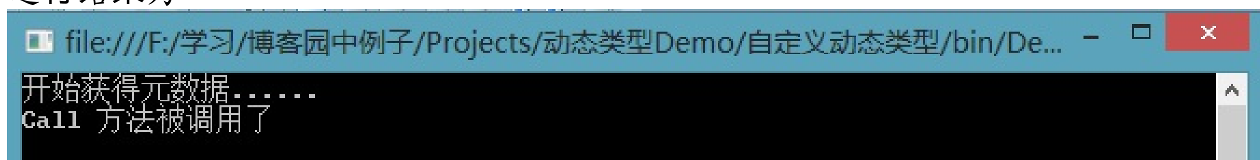
6.3 实现IDynamicMetaObjectProvider接口来实现动态行为

由于Dynamic类型在运行时来动态创建对象的，所以对该类型的每个成员的访问都会调用GetMetaObject方法来获得动态对象，然后通过这个动态对象来进行调用，所以实现IDynamicMetaObjectProvider接口，需要实现一个GetMetaObject方法来返回DynamicMetaObject对象，演示代码如下：

```
 static void Main(string[] args)
        {
            dynamic dynamicobj2 = new DynamicType2();
            dynamicobj2.Call();
            Console.Read();
 }
 public class DynamicType2 : IDynamicMetaObjectProvider
     {
         public DynamicMetaObject GetMetaObject(Expression paramete
         {
             Console.WriteLine("开始获得元数据......");
             return new Metadynamic(parameter,this);
         }
     }

     // 自定义Metadynamic类
     public class Metadynamic : DynamicMetaObject
     {
         internal Metadynamic(Expression expression, DynamicType2 va
             : base(expression, BindingRestrictions.Empty, value)
         {
         }
         // 重写响应成员调用方法
         public override DynamicMetaObject BindInvokeMember(InvokeMe
         {
             // 获得真正的对象
             DynamicType2 target = (DynamicType2)base.Value;
             Expression self = Expression.Convert(base.Expression, t
             var restrictions = BindingRestrictions.GetInstanceRestr
             // 输出绑定方法名
             Console.WriteLine(binder.Name + " 方法被调用了");
             return new DynamicMetaObject(self, restrictions);
         }
     }
```

运行结果为：



七、总结

讲到这里动态类型的介绍就已经介绍完了，本专题差不多涵盖了动态类型中所有内容，希望通过本专题大家能够对C# 4.0中提出来的动态类型特性可以有进一步的了解，并且本专题也是这个系列中的最后一篇文章了，到这里C#基础知识系列也就结束了，后面我会整理出这个系列文章的一个索引，从而方便大家收藏，然而C#4中对COM互操作性也有很大的改善，关于互操作的内容将会在后面一个系列文章中和大家分享下我的学习体会。眼看都2点20了，该睡觉去了。

[你必须知道的异步编程]C# 5.0 新特性——Async和Await使异步编程更简单

本专题概要:

一、引言

在之前的C#基础知识系列文章中只介绍了从C#1.0到C#4.0中主要的特性，然而.NET 4.5 的推出，对于C#又有了新特性的增加——就是C#5.0中async和await两个关键字，这两个关键字简化了异步编程，之所以简化了，还是因为编译器给我们做了更多的工作，下面就具体看看编译器到底在背后帮我们做了哪些复杂的工作的。

二、同步代码存在的问题

对于同步的代码，大家肯定都不陌生，因为我们平常写的代码大部分都是同步的，然而同步代码却存在一个很严重的问题，例如我们向一个Web服务器发出一个请求时，如果我们发出请求的代码是同步实现的话，这时候我们的应用程序就会处于等待状态，直到收回一个响应信息为止，然而在这个等待的状态，对于用户不能操作任何的UI界面以及也没有任何的消息，如果我们试图去操作界面时，此时我们就会看到"应用程序为响应"的信息(在应用程序的窗口旁)，相信大家在平常使用桌面软件或者访问web的时候，肯定都遇到过这样类似的情况的，对于这个，大家肯定会觉得看上去非常不舒服。引起这个原因正是因为代码的实现是同步实现的，所以在没有得到一个响应消息之前，界面就成了一个"卡死"状态了，所以这对于用户来说肯定是不可接受的，因为如果我要从服务器上下载一个很大的文件时，此时我们甚至不能对窗体进行关闭的操作的。为了具体说明同步代码存在的问题(造成界面开始)，下面通过一个程序让大家更形象地看下问题所在：

```
// 单击事件
        private void btnClick_Click(object sender, EventArgs e)
        {
            this.btnClick.Enabled = false;

            long length = AccessWeb();
            this.btnClick.Enabled = true;
            // 这里可以做一些不依赖回复的操作
            OtherWork();

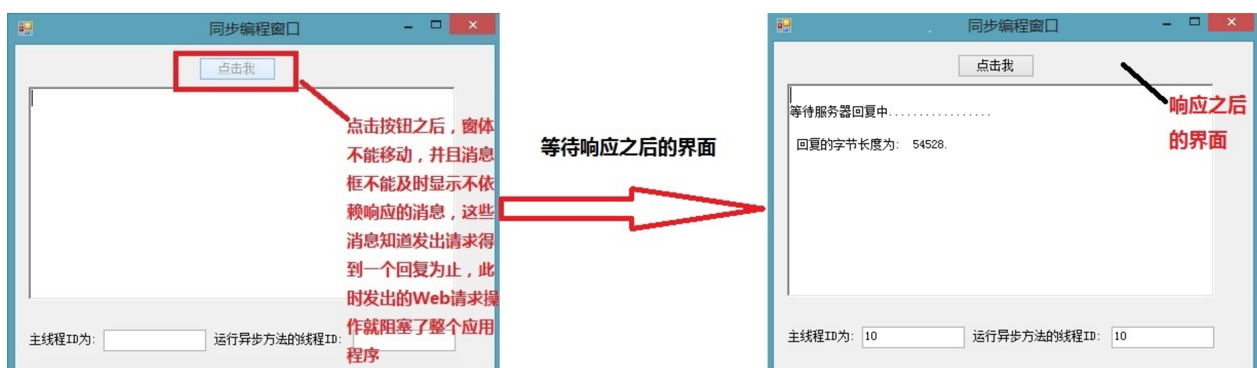
            this.richTextBox1.Text += String.Format("\n 回复的字节长
            txbMainThreadID.Text = Thread.CurrentThread.ManagedThre
        }

        private  long AccessWeb()
        {
            MemoryStream content = new MemoryStream();

            // 对MSDN发起一个Web请求
            HttpWebRequest webRequest = WebRequest.Create("http://
            if (webRequest != null)
            {
                // 返回回复结果
                using (WebResponse response = webRequest.GetRespons
                {
                    using (Stream responseStream = response.GetResp
                    {
                        responseStream.CopyTo(content);
                    }
                }
            }

            txbAsynMethodID.Text = Thread.CurrentThread.ManagedThre
            return content.Length;
        }
```

运行程序后，当我们点击窗体的 "点击我"按钮之后，在得到服务器响应之前，我们不能对窗体进行任何的操作，包括移动窗体，关闭窗体等，具体运行结果如下：



三、传统的异步编程来改善程序的响应

上面部分我们已经看到同步方法所带来的实际问题了，为了解决类似的问题，.NET Framework很早就提供了对异步编程的支持，下面就用.NET 1.0中提出的异步编程模型(APM)来解决上面的问题，具体代码如下(注释的部分通过获得GUI线程的同步上文对象，然后同步调用同步上下文对象的post方法把要调用的方法交给GUI线程去处理，因为控件本来就是由GUI线程创建的，然后由它自己执行访问控件的操作就不存在跨线程的问题了，程序中使用的是调用**RichTextBox**控件的**Invoke**方式来异步回调访问控件的方法，其实背后的原来和注释部分是一样的，调用**RichTextBox**控件的**Invoke**方法可以获得创建RichTextBox控件的线程信息(也就是前一种方式的同步上下文)，然后让Invoke回调的方法在该线程上运行):

```
private void btnClick_Click(object sender, EventArgs e)
        {
            this.richTextBox1.Clear();
            btnClick.Enabled = false;
            AsyncMethodCaller caller = new AsyncMethodCaller(TestMe
            IAsyncResult result = caller.BeginInvoke(GetResult, nul

            //// 捕捉调用线程的同步上下文派生对象
            //sc= SynchronizationContext.Current;
        }

        # region 使用APM实现异步编程
        // 同步方法
        private string TestMethod()
        {
            // 模拟做一些耗时的操作
            // 实际项目中可能是读取一个大文件或者从远程服务器中获取数据等。
            for (int i = 0; i < 10; i++)
            {
                Thread.Sleep(200);
            }

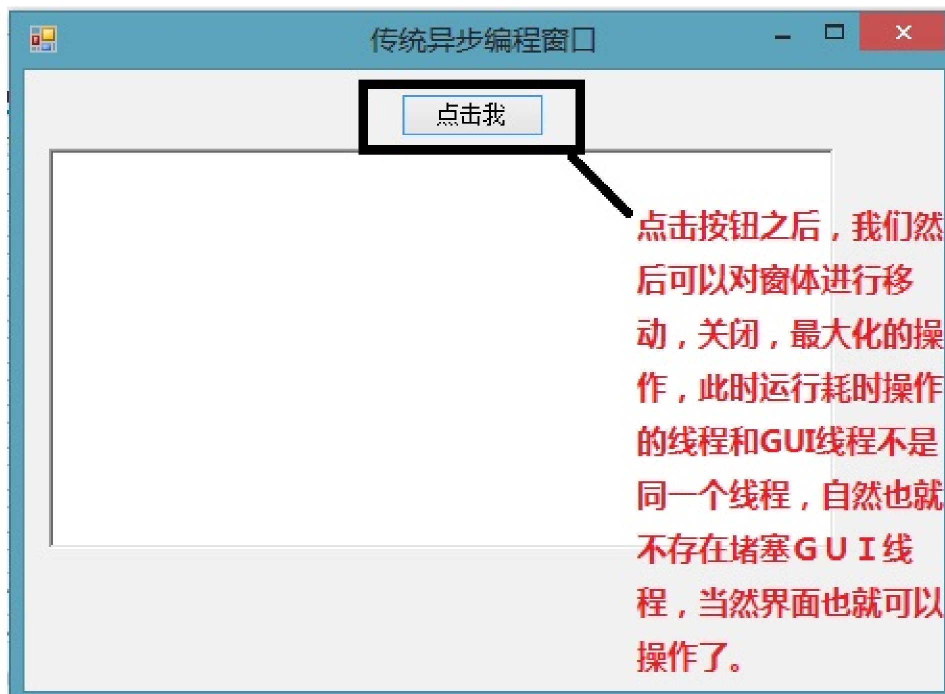
            return "点击我按钮事件完成";
        }

        // 回调方法
        private void GetResult(IAsyncResult result)
        {
            AsyncMethodCaller caller = (AsyncMethodCaller)((AsyncRe
            // 调用EndInvoke去等待异步调用完成并且获得返回值
            // 如果异步调用尚未完成，则 EndInvoke 会一直阻止调用线程，直到
            string resultvalue = caller.EndInvoke(result);
            //sc.Post(ShowState,resultvalue);
            richTextBox1.Invoke(showStateCallback, resultvalue);
        }

        // 显示结果到richTextBox
        private void ShowState(object result)
        {
            richTextBox1.Text = result.ToString();
            btnClick.Enabled = true;
        }

        // 显示结果到richTextBox
        //private void ShowState(string result)
        //{
        //    richTextBox1.Text = result;
        //    btnClick.Enabled = true;
        //}
        #endregion
```

运行的结果为：



四、C# 5.0 提供的async和await使异步编程更简单

上面部分演示了使用传统的异步编程模型(APM)来解决同步代码所存在的问题，然而在.NET 2.0，.NET 4.0和.NET 4.5中，微软都有推出新的方式来解决同步代码的问题，他们分别为基于事件的异步模式，基于任务的异步模式和提供async和await关键字来对异步编程支持。关于前两种异步编程模式，在我前面的文章中都有介绍，大家可以查看相关文章进行详细地了解，本部分就C# 5.0中的async和await这两个关键字如何实现异步编程的问题来给大家介绍下。下面通过代码来了解下如何使用async和await关键字来实现异步编程，并且大家也可以参看前面的博客来对比理解使用async和await是异步编程更简单。

```
private async void btnClick_Click(object sender, EventArgs e)
        {
            long length = await AccessWebAsync();

            // 这里可以做一些不依赖回复的操作
            OtherWork();

            this.richTextBox1.Text += String.Format("\n 回复的字节长
            txbMainThreadID.Text = Thread.CurrentThread.ManagedThre
        }

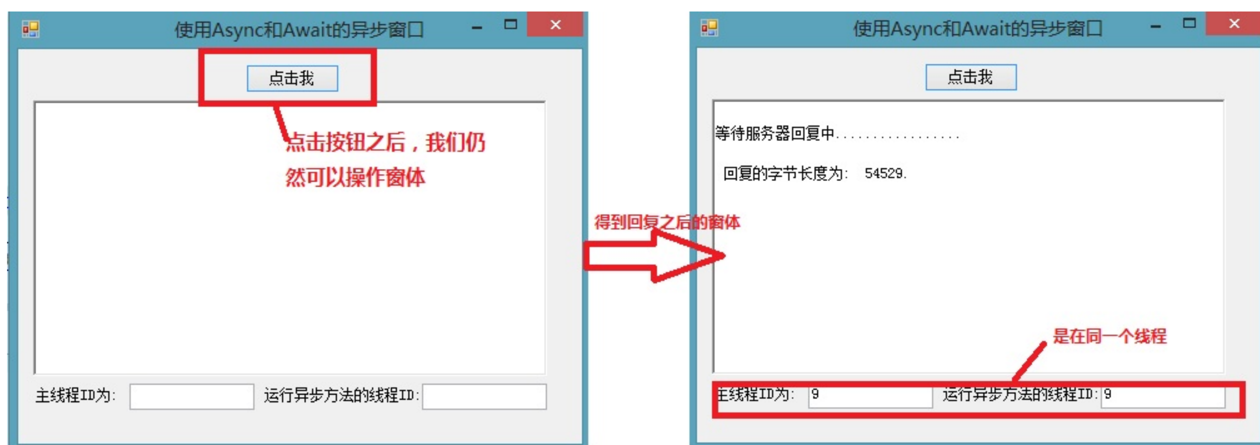
        // 使用C# 5.0中提供的async 和await关键字来定义异步方法
        // 从代码中可以看出C#5.0 中定义异步方法就像定义同步方法一样简单。
        // 使用async 和await定义异步方法不会创建新线程,
        // 它运行在现有线程上执行多个任务.
        // 此时不知道大家有没有一个疑问的？在现有线程上(即UI线程上)运行一个耗
        // 为什么不会堵塞UI线程的呢？
        // 这个问题的答案就是 当编译器看到await关键字时，线程会
        private async Task<long> AccessWebAsync()
        {
            MemoryStream content = new MemoryStream();

            // 对MSDN发起一个Web请求
            HttpWebRequest webRequest = WebRequest.Create("http://m
            if (webRequest != null)
            {
                // 返回回复结果
                using (WebResponse response = await webRequest.GetR
                {
                    using (Stream responseStream = response.GetResp
                    {
                        await responseStream.CopyToAsync(content);
                    }
                }
            }

            txbAsynMethodID.Text = Thread.CurrentThread.ManagedThre
            return content.Length;
        }

        private void OtherWork()
        {
            this.richTextBox1.Text += "\r\n等待服务器回复中..........
        }
```

运行结果如下：



五、async和await关键字剖析

我们对比下上面使用async和await关键字来实现异步编程的代码和在第二部分的同步代码，有没有发现使用async和await关键字的异步实现和同步代码的实现很像，只是异步实现中多了async和await关键字和调用的方法都多了async后缀而已。正是因为他们的实现很像，所以我在第四部分才命名为使用async和await使异步编程更简单，就像我们在写同步代码一样，并且代码的coding思路也是和同步代码一样，这样就避免考虑在APM中委托的回调等复杂的问题，以及在EAP中考虑各种事件的定义。从代码部分我们可以看出async和await的使用确实很简单，我们就如在写同步代码一般，但是我很想知道编译器到底给我们做了怎样的处理的？并且从运行结果可以发现，运行异步方法的线程和GUI线程的ID是一样的，也就是说异步方法的运行在GUI线程上，所以就不用像APM中那样考虑跨线程访问的问题了(因为通过委托的BeginInvoke方法来进行回调方法时，回调方法是在线程池线程上执行的)。下面就用反射工具看看编译器把我们的源码编译成什么样子的：

对于按钮点击事件的代码来说，编译器生成的背后代码却是下面这样的，完全和我们源码中的两个样：

```
// 编译器为按钮Click事件生成的代码
private void btnClick_Click(object sender, EventArgs e)
{
    <btnClick_Click>d__0 d__;
    d__.<>4__this = this;
    d__.sender = sender;
    d__.e = e;
    d__.<>t__builder = AsyncVoidMethodBuilder.Create();
    d__.<>1__state = -1;
    d__.<>t__builder.Start<<btnClick_Click>d__0>(ref d__);
}
```

看到上面的代码,作为程序员的我想说——编译器你怎么可以这样呢？怎么可以任意篡改我的代码呢？这样不是侵犯我的版权了吗？你要改最起码应该告诉我一声吧，如果我的源码看到它在编译器中的实现是上面那样的，我相信我的源码会说——难

道我中了世间上最恶毒的面目全非脚吗？ 好吧，为了让大家更好地理清编译器背后到底做了什么事情，下面就顺着上面的代码摸瓜，我也来展示耍一套还我漂漂拳来帮助大家找到编译器代码和源码的对应关系。我的分析思路为:

1、提出问题——我的click事件的源码到哪里去了呢？

从编译器代码我们可以看到，前面的7句代码都是对某个类进行赋值的操作，最真正起作用的就是最后Start方法的调用。这里又产生了几个疑问——<btnClick_Click>d**0**是什么类型**?** 该类型中的**<>t**builder字段类型的Start方法到底是做什么用的？ 有了这两个疑问，我们就点击<btnClick_Click>d__0(反射工具可以让我们直接点击查看)来看看它是什么类型

```
// <btnClick_Click>d__0类型的定义，从下面代码可以看出它是一个结构体
// 该类型是编译器生成的一个嵌入类型
// 看到该类型的实现有没有让你联想到什么？
private struct <btnClick_Click>d__0 : IAsyncStateMachine
{
    // Fields
    public int <>1__state;
    public Form1 <>4__this;
    public AsyncVoidMethodBuilder <>t__builder;
    private object <>t__stack;
    private TaskAwaiter<long> <>u__$awaiter2;
    public long <length>5__1;
    public EventArgs e;
    public object sender;

    // Methods
    private void MoveNext()
    {
        try
        {
            TaskAwaiter<long> CS$0$0001;
            bool <>t__doFinallyBodies = true;
            switch (this.<>1__state)
            {
                case -3:
                    goto Label_010E;

                case 0:
                    break;

                default:
// 获取用于等待Task（任务）的等待者。你要知道某个任务是否完成，我们就需要一个等
// 从这里可以看出，其实async和await关键字背后的实现原理是基于任务的异步编程模
                   ** // 这里代码是在线程池线程上运行的**
                    CS$0$0001 = this.<>4__this.AccessWebAsync().Get
// 如果任务完成就调转到Label_007A部分的代码
                    if (CS$0$0001.IsCompleted)
                    {
                        goto Label_007A;
                    }

                    // 设置状态为0为了退出回调方法。
                    this.<>1__state = 0;
                    this.<>u__$awaiter2 = CS$0$0001;
// 这个代码是做什么用的呢？让我们带着问题看下面的分析
```

```
this.<>t__builder.AwaitUnsafeOnCompleted<TaskAwaiter<long>, Form1.<
                    <>t__doFinallyBodies = false;
// 返回到调用线程,即GUI线程,这也是该方法不会堵塞GUI线程的原因,不管任务是否完成
                    return;
                }
                // 当任务完成时，不会执行下面的代码，会直接执行Label_007A中代码
                CS$0$0001 = this.<>u__$awaiter2;
                this.<>u__$awaiter2 = new TaskAwaiter<long>();
                // 为了使再次回调MoveNext代码
                this.<>1__state = -1;
            Label_007A:
                **// 下面代码是在GUI线程上执行的**
                CS$0$0001 = new TaskAwaiter<long>();
                long CS$0$0003 = CS$0$0001.GetResult();
                this.<length>5__1 = CS$0$0003;
// 我们源码中的代码这里的
                **this.<>4__this.OtherWork();
                this.<>4__this.richTextBox1.Text = this.<>4__this.richT
                this.<>4__this.txbMainThreadID.Text =** **Thread.Curren
            }
            catch (Exception <>t__ex)
            {
                this.<>1__state = -2;
                this.<>t__builder.SetException(<>t__ex);
                return;
            }
        Label_010E:
            this.<>1__state = -2;
            this.<>t__builder.SetResult();
        }

        [DebuggerHidden]
        private void SetStateMachine(IAsyncStateMachine param0)
        {
            this.<>t__builder.SetStateMachine(param0);
        }
}
```

如果你看过我的迭代器专题的话,相信你肯定可以联想到该结构体就是一个迭代器的一个实现,其主要方法就是MoveNext方法。从上面的代码的注释应该可以帮助我们解决在第一步提到的第一个问题,即<btnClick_Click>d0是什么类型,下面就分析下第二个问题，从**<btnClick_Click>d**0结构体的代码中可以发现<>t__builder的类型是**AsyncVoidMethodBuilder**类型,下面就看看它的**Start**方法的解释——运行关联状态机的生成器,即调用该方法就可以开始运行状态机,运行状态机指的就是执行**MoveNext**方法(**MoveNext**方法中有我们源码中所有代码,这样就把编译器生成的**Click**方法与我们的源码关联起来了)。从上面代码注释中可以发现，当该**MoveNext**被调用时会立即还回到**GUI**线程中，同时也有这样的疑问——刚开始调用**MoveNext**方法时，任务肯定是还没有被完成的，但是我们输出我们源码中的代码，必须等待任务完成（因为任务完成才能调转到**Label_007A**中的代码），此时

我们应该需要回调**MoveNext**方法来检查任务是否完成，**(**就如迭代器中的，我们需要使用**foreach**语句一直调用**MoveNext**方法**)**，然而我们在代码却没有找到回调的任何代码啊？ 对于这个疑问，回调**MoveNext**方法肯定是存在的，只是首次看上面代码的朋友还没有找到类似的语句而已，上面代码注释中我提到了一个问题**——**这个代码是做什么用的呢？让我们带着问题看下面的分析,其实注释下面的代码就是起到回调**MoveNext**方法的作用，**AsyncVoidMethodBuilder.AwaitUnsafeOnCompleted<TAwaiter, TStateMachine>** 方法就是调度状态机去执行**MoveNext**方法**,**从而也就解决了回调**MoveNext**的疑问了。

相信大家从上面的解释中可以找到源码与编译器代码之间的对应关系了吧, 但是我在分析完上面的之后,又有一个疑问——当任务完成时，是如何退出MoveNext方法的呢？总不能让其一直回调吧，从上面的代码的注释可以看出，当任务执行完成之后，会把<>1**state**设置为**0,**当下次再回调**MoveNext**方法时就会直接退出方法，然而任务没完成之前，同样也会把**<>1**state设置为0，但是Switch部分后面的代码又把<>1**state**设置为**-1,**这样就保证了在任务没完成之前，**MoveNext**方法可以被重复回调，当任务完成之后，**<>1**state设置为-1的代码将不会执行，而是调转到Label_007A部分。

经过上面的分析之后,相信大家也可以耍出一套还我漂漂拳去分析异步方法**AccessWebAsync**(),其分析思路是和btnClick_Click的分析思路是一样的.这里就不重复啰嗦了。

分析完之后,下面再分享下几个关于async和await常问的问题

问题一：是不是写了async关键字的方法就代表该方法是异步方法，不会堵塞线程呢？

答: 不是的， 对于只标识async关键字的(指在方法内没有出现await关键字)的方法,调用线程会把该方法当成同步方法一样执行,所以然而会堵塞GUI线程,只有当async和await关键字同时出现，该方法才被转换为异步方法处理。

问题二：“async”关键字会导致调用方法用线程池线程运行吗?

答: 不会,被async关键字标识的方法不会影响方法是同步还是异步运行并完成,而是，它使方法可被分割成多个片段，其中一些片段可能异步运行，这样这个方法可能异步完成。这些片段界限就出现在方法内部显示使用”await”关键字的位置处。所以，如果在标记了”async”的方法中没有显示使用”await”，那么该方法只有一个片段，并且将以同步方式运行并完成。在await关键字出现的前面部分代码和后面部分代码都是同步执行的(即在调用线程上执行的，也就是GUI线程，所以不存在跨线程访问控件的问题)，await关键处的代码片段是在线程池线程上执行。总结为——使用async和await关键字实现的异步方法，此时的异步方法被分成了多个代码片段去执行的，而不是像之前的异步编程模型(APM)和EAP那样，使用线程池线程去执行一整个方法。

关于更多async和await关键字的常问问题可以查看——Async/Await FAQ和中文翻译——（译）关于async与await的FAQ

六、小结

写到这里本专题的内容就介绍到这里的,并且我也会把本专题的内容同步到之前的C#基础知识系列文章索引，这样我的C#特性系列也就完整了，并且该专题也是异步编程的最后一篇专题，在后面的专题将为大家实现一个类似迅雷的多任务多线程下载器，对于这个专题可能会用到并行编程的内容，所以接下面我为为大家分享下并行编程的内容。

根据 一路转圈的雪人的建议，因为对于刚使用await的人，经常会问“帮来看一下怎么死锁了，怎么办啊，要死了，怎么解决？”，对于这样的问题大家应该明白一点就是——使用**async**标识的异步方法的运行在**GUI**线程上**(**对于这点大家一定要明白，在我文章中的剖析部分也详细介绍了原因，阅读文章的人应该重点了解**)**，所以就不用像**APM**中那样考虑跨线程访问的问题了。

本专题所有源码下载：ASyncAndAwaitTestProject.zip

全面解析C#中参数传递

一、引言

对于一些初学者(包括工作几年的人在内)来说，有时候对于方法之间的参数传递的问题感觉比较困惑的，因为之前在面试的过程也经常遇到参数传递的基础面试题，这样的面试题主要考察的开发人员基础是否扎实，对于C#中值类型和引用类型有没有深入的一个理解——这个说的理解并不是简单的对它们简单一个定义描述，而在于它们在内存中分布。所以本文章将带领大家深入剖析下C#中参数传递的问题，并分享我自己的一个理解，只有你深入理解了才能在不运行程序的情况就可以分析出参数传递的结果的。

二、按值传递

对于C#中的参数传递，根据参数的类型可以分为四类：

- 值类型参数的按值传递
- 引用类型参数的按值传递
- 值类型参数的按引用传递
- 引用类型参数的按引用传递

然而在默认情况下，CLR方法中参数的传递都是按值传递的。为了帮助大家全面理解参数的传递，下面就这四种情况一一进行分析。

2.1 值类型参数的按值传递

对于参数又分为:形参和实参,形参指的是被调用方法中的参数,实参指的是调用方法的参数,下面结合代码帮助大家理解形参和实参的概念：

```
class Program
    {
        static void Main(string[] args)
        {

            int addNum = 1;
            // addNum 就是实参,
            Add(addNum);
         }

        // addnum就是形参，也就是被调用方法中的参数
        private static void Add(int addnum)
        {
            addnum = addnum + 1;
            Console.WriteLine(addnum);
        }
}
```

对于值类型的按值传递,传递的是该值类型实例的一个拷贝,也就是形参此时接受到的是实参的一个副本,被调用方法操作是实参的一个拷贝,所以此时并不影响原来调用方法中的参数值,为了证明这点,看看下面的代码和运行结果就明白了:

```
 class Program
    {
        static void Main(string[] args)
        {
            // 1\. 值类型按值传递情况
            Console.WriteLine("按值传递的情况");
            int addNum = 1;
            Add(addNum);
            Console.WriteLine(addNum);

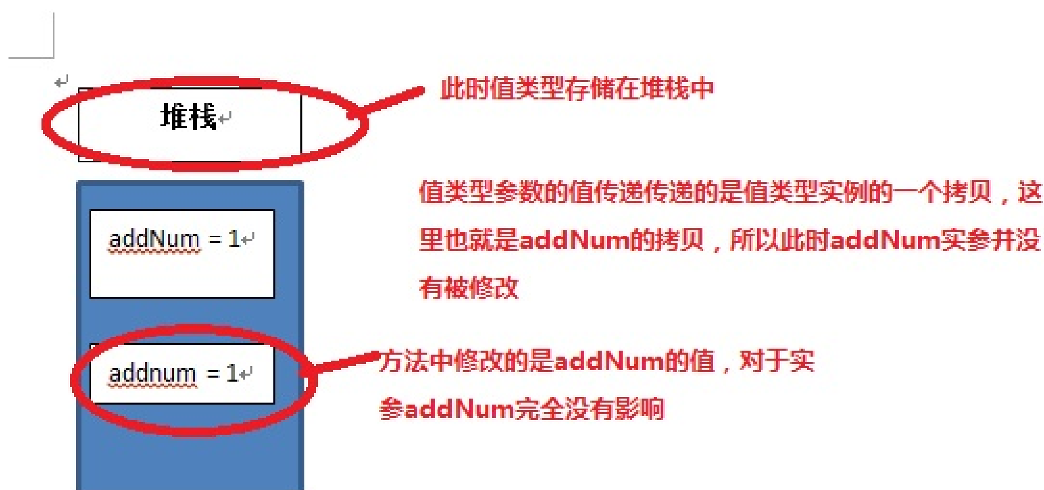
            Console.Read();
        }

        // 1\. 值类型按值传递情况
        private static void Add(int addnum)
        {
            addnum = addnum + 1;
            Console.WriteLine(addnum);
        }
```

运行结果为:



从结果中可以看出addNum调用方法之后它的值并没有改变，Add 方法的调用只是改变了addNum的副本addnum的值，所以addnum的值修改为2了。然而我们的分析到这里并没有结束，为了让大家深入理解传递传递，我们有必要知道为什么值类型参数的按值传递不会修改实参的值，相信下面这张图可以解释你所有的疑惑：

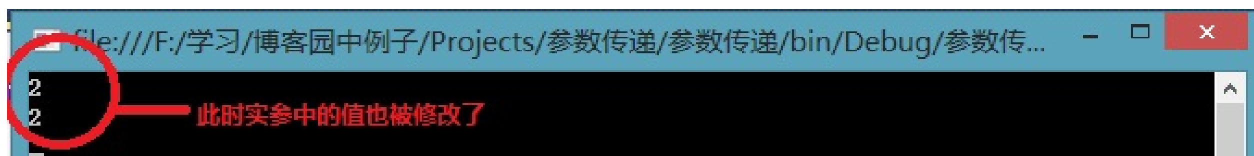


2.2 引用类型参数的按值传递

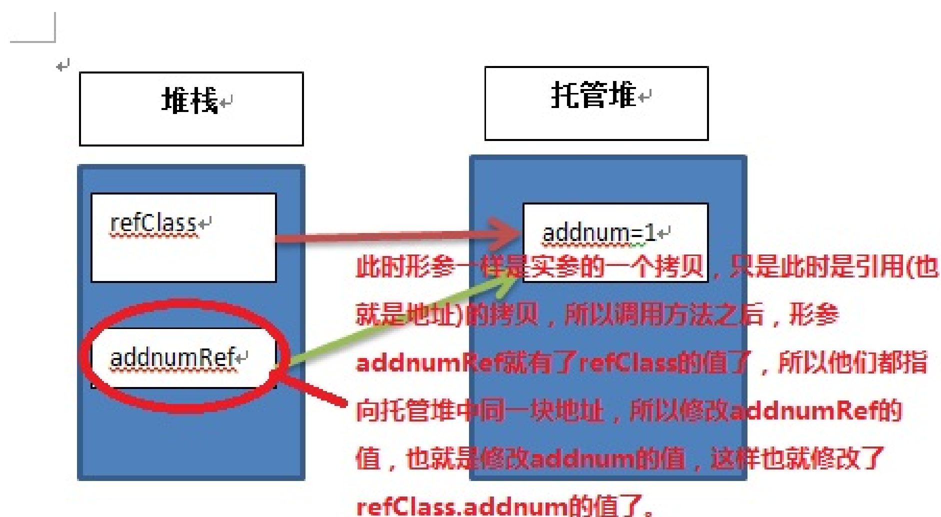
当传递的参数是引用类型的时候，传递和操作的是指向对象的引用（看到这里，有些朋友会觉得此时不是传递引用吗？怎么还是按值传递了？对于这个疑惑，此时确实是按值传递，此时传递的对象的地址，传递地址本身也是传递这个地址的值，所以此时仍然是按值传递的），此时方法的操作就会改变原来的对象。对于这点可能看文字描述会比较难理解下面结合代码和分析图来帮助大家理解下：

```
class Program
    {
        static void Main(string[] args)
        {
            // 2\. 引用类型按值传递情况
            RefClass refClass = new RefClass();
            AddRef(refClass);
            Console.WriteLine(refClass.addnum);
        }
         // 2\. 引用类型按值传递情况
        private static void AddRef(RefClass addnumRef)
        {
            addnumRef.addnum += 1;
            Console.WriteLine(addnumRef.addnum);
        }
}
class RefClass
    {
        public int addnum=1;
    }
```

运行结果为：



为什么此时传递引用就会修改原来实参中的值呢？对于这点我们还是参数在内存中分布图来解释下：



2.3 .String引用类型的按值传递的特殊情况

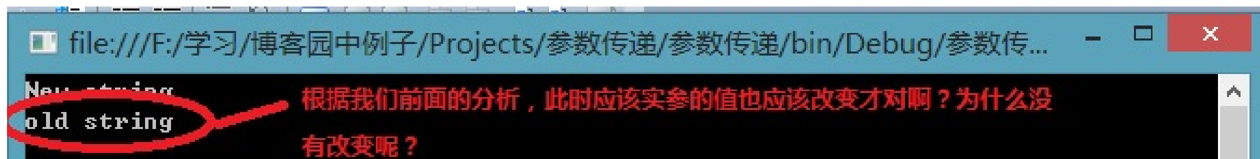
对于String类型同样是引用类型，然而对于string类型的按值传递时，此时引用类型的按值传递却不会修改实参的值，可能很多朋友对于这点很困惑，下面具体看看下面的代码：

```
class Program
    {
        static void Main(string[] args)
        {
              // 3\. String引用类型的按值传递的特殊情况
            string str = "old string";
            ChangeStr(str);
            Console.WriteLine(str);

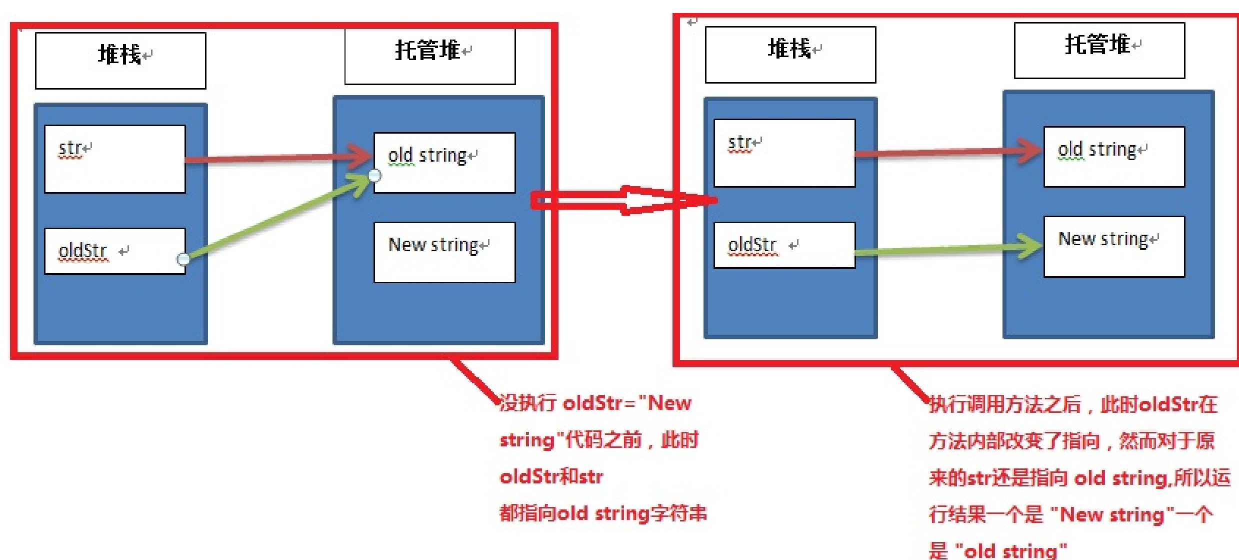
        }

         // 3\. String引用类型的按值传递的特殊情况
        private static void ChangeStr(string oldStr)
        {
            oldStr = "New string";
            Console.WriteLine(oldStr);
        }
}
```

运行结果为：



对于为什么原来的值没有被改变主要是因为string的“不变性”，所以在被调用方法中执行 **oldStr="New string"**代码时，此时并不会直接修改**oldStr**中的**"old string"**值为**"New string"**，因为**string**类型是不变的，不可修改的，此时内存会重新分配一块内存，然后把这块内存中的值修改为 **“New string”**，然后把内存中地址赋值给**oldStr**变量，所以此时**str**仍然指向 **"old string"**字符，而**oldStr**却改变了指向，它最后指向了 **"New string"**字符串。所以运行结果才会像上面这样，下面内存分布图可以帮助你更形象地理解文字表述：



三、按引用传递

不管是值类型还是引用类型，我们都可以使用ref 或out关键字来实现参数的按引用传递，然而按引用进行传递的时候，需要注意下面两点：

方法的定义和方法调用都必须同时显式使用ref或out,否则会出现编译错误

CLR允许通过out 或ref参数来实现方法重载。如：

```
    #region CLR 允许out或ref参数来实现方法重载
        private static void Add(string str)
        {
            Console.WriteLine(str);
        }

        // 编译器会认为下面的方法是另一个方法，从而实现方法重载
        private static void Add(ref string str)
        {
            Console.WriteLine(str);
        }
        #endregion
```

按引用传递可以解决由于值传递时改变引用副本而不影响引用本身的问题，此时传递的是引用的引用（也就是地址的地址），而不是引用的拷贝（副本）。下面就具体看看按引用传递的代码：

```
class Program
    {
        static void Main(string[] args)
        {
            #region 按引用传递
            Console.WriteLine("按引用传递的情况");
            int num = 1;
            string refStr = "Old string";
            ChangeByValue(ref num);
            Console.WriteLine(num);
            changeByRef(ref refStr);
            Console.WriteLine(refStr);
            #endregion

            Console.Read();
        }

        #region 按引用传递
        // 1\. 值类型的按引用传递情况
        private static void ChangeByValue(ref int numValue)
        {
            numValue = 10;
            Console.WriteLine(numValue);
        }

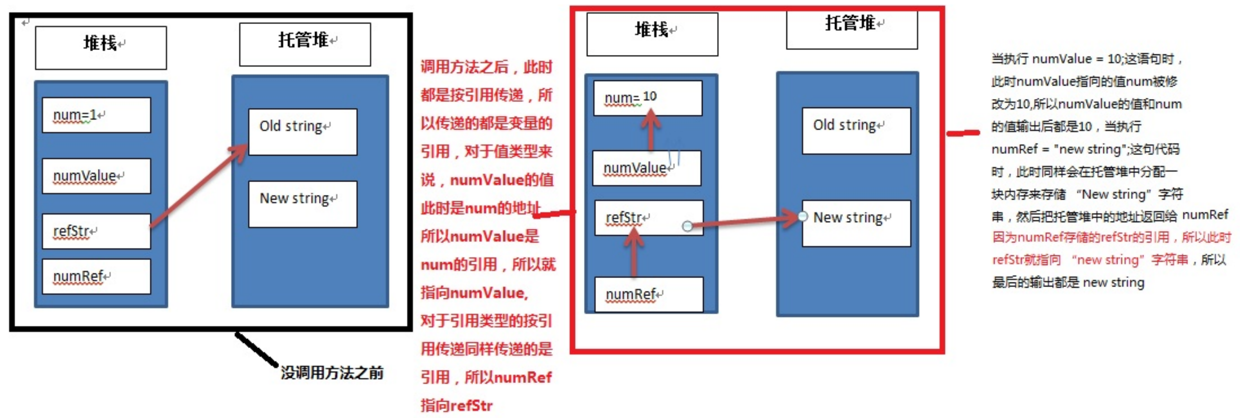
        // 2\. 引用类型的按引用传递情况
        private static void changeByRef(ref string numRef)
        {
            numRef = "new string";
            Console.WriteLine(numRef);
        }

        #endregion
}
```

运行结果为：

```
file:///F:/学习/博客园中例子/Projects/参数传递/参数传递/bin/Debug/参数传...
按引用传递的情况
10
10
new string
new string
```

从运行结果可以看出，此时引用本身的值也被改变了，通过下面一张图来帮忙大家理解下按引用传递的方式：



四、总结

到这里参数的传递所有内容就介绍完了。总之，对于按值传递，不管是值类型还是引用类型的按值传递，都是传递实参的一个拷贝，只是值类型时，此时传递的是实参实例的一个拷贝（也就是值类型值的一个拷贝），而引用类型时，此时传递的实参引用的副本。对于按引用传递，传递的都是参数地址，也就是实例的指针。

所有源码下载：参数传递

[C#基础知识系列]全面解析C#中静态与非静态

一、引言

在C#中,静态和非静态的特征对于我们来说是再熟悉不过了,但是很少看到有一篇文章去好好地总结静态和非静态它们之间的不同,为了帮助大家更好地去理解静态和非静态特征, 所以将在这篇文章中帮大家全面总结下它们之间的不同,包括静态类,静态成员和静态构造函数。希望在大家巩固基础的时候可以拿出来好好复习下的。下面废话不多了，直接进入我们今天的主题。

二、为什么需要静态特征

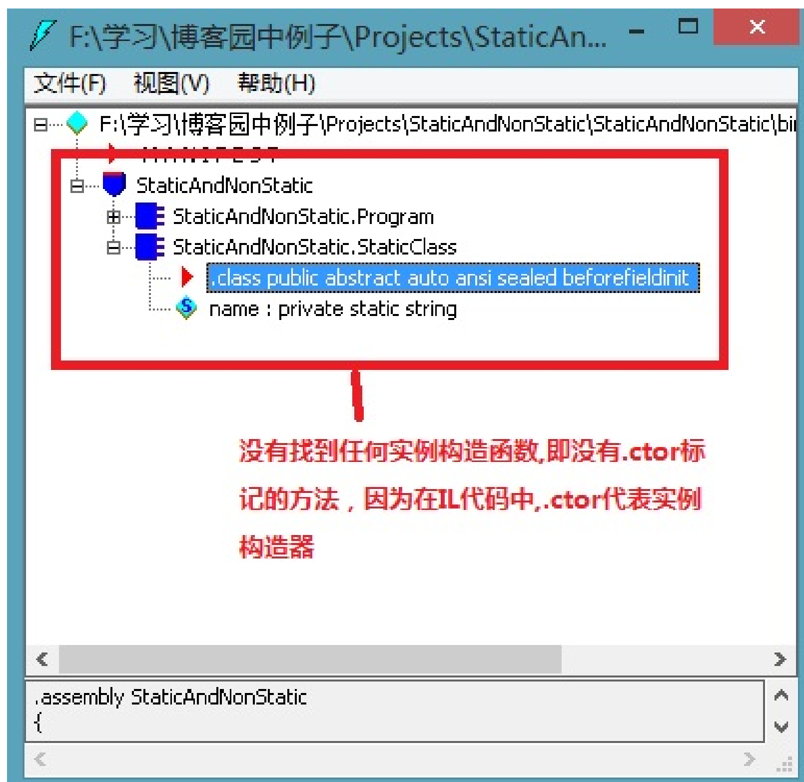
在自定义类或看.NET Framework类库中都可以发现，类中大部分都是具体实例特征（也就是没有static标识的），同时我们也能看到一些具有静态特征的类或成员，例如我们经常使用的Console类以及WriteLine方法就是静态的。然而有些朋友会疑惑,为什么还要有静态特征的呢?干脆都定义为实例的好了? 然后静态特征的存在肯定有它存在的原因的,并不是我们就是要这么定义的,其实我一直认为不管是什么都是源于生活的, 技术的实现也是一样,比如我们开发程序,需要掌握技术外,其实更重要的是业务逻辑这块的,如果你都不知道你开发的东西是怎样的一个流程,即使你技术再牛做出来的东西都是反人类的东西(也就是指不符合用户的用户习惯和之前的一个业务需求),其实静态特征的存在也是源于生活的,对于类好比就是我们现实生活中的人或事物,静态特征和非静态特征就好比生活中人或事物具有的特征, 我们询问人的时候或者电视剧警察查案件的时候,都会听到这样一句话 "那个人有什么特征？"或 “嫌疑犯有什么特征？多高，年龄等” 其实高度、年龄、性别都是一个人的特征，所以这些在语言范畴就需要为其进行定义了,也就是我们定义的实例成员了,然而有些特征需要被所有对象实例所共有的,这些特征在语言范畴就定义为静态特征,具体哪些特征可以定义为静态特征呢? 其实这点一样是源于生活的,所以我们在开发软件的过程中,必不可少的一个流程就是需求分析了,只有在了解客户需求的条件下才能进行之后的所有流程的, 例如一个班级有很多学生,每个学生是一个实体,在语言范畴就可以定义一个类,当我们需要一个学生的时候就可以通过new 关键字创建一个出来（说到这里又让我想到了恶搞泰囧的图片——你有对象吗？没对象，你们程序员可以自己new一个啊？）,然而我们创建出来的学生他们都有一些共有的特征,如同一个班级,学校等, 如果我们把班级、学校这样的特征也定义为实例的话，那么我们不是每次创建对象实例的时候都为这些共有的特征分配一次内存的,这样不仅对内存空间的浪费也是不满足生活常识的,此时我们就可以把班级、学校这样的特征定义为静态特征，这样所有实例都可以共享这两个特征，并且不需要为每个对象实例分配内存。

三、比较静态特征和非静态特征

3.1 静态类与非静态类

- 静态类和非静态类在C#中定义基本是一样的,只是静态类定义需要加上static修饰符而已。下面就直接总结下它们之间的区别：
- 静态类只能包含静态成员，否则会抛出编译错误；然而非静态类既可以包含非静态成员也可以包含静态成员
- 静态类是不能实例化,之所以不能实例化,是因为静态类会导致C#编译器将该类同时标记为abstract和sealed,并且编译器不会在类型中生成一个实例的构造函数，从而导致静态类不能实例化，具体原因可以见下图；非静态类可以，并且静态成员的访问只能通过类来进行访问，因为静态成员是属于类的。

上面代码用IL反汇编程序得到的IL代码结构为:

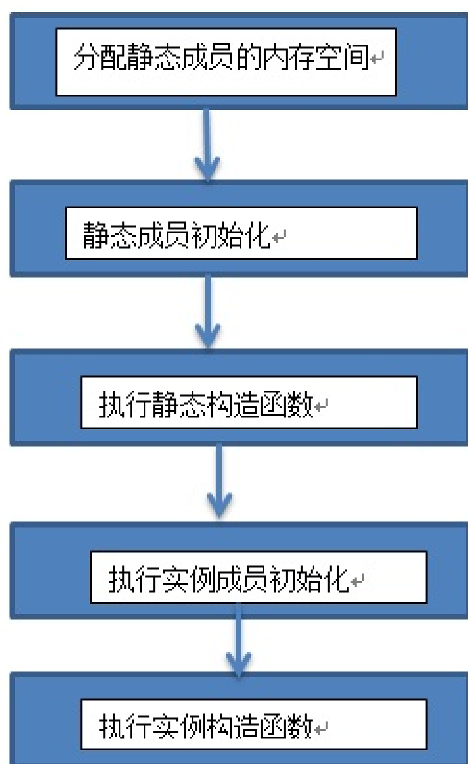


3.2 静态构造函数与实例构造函数

- 静态构造函数用来初始化类中的静态成员的,包括静态字段和静态属性,并且静态构造函数是不能带有参数、不能有访问修饰符，静态构造函数的调用是由CLR第一次调用类成员之前执行的。
- 下面还是直接总结下静态构造函数与实例构造函数之间的区别:
- 静态构造函数可以与无参的实例构造函数同时存在
- 静态构造函数在CLR加载类时执行,然而实例构造函数在每次实例创建时都会执行
- 静态构造函数只能对静态成员初始化,不能对非静态成员进行初始化操作,然而实例构造函数,既可以初始化实例成员也可以初始化静态成员,但静态只读字段除外
- 静态构造函数只被执行一次,但是CLR也不能确定它什么时候被执行,它的执行方式有两种,precise和before-field-init,这个会在下一篇文章中详细给大家介绍，这里先提出给大家一个思考的空间。而实例构造函数在每次创建对象实例时都会被执行，创建几个就会执行几次

- 一个类只能有一个静态构造函数，却可以有多个实例构造函数

静态字段的初始值在静态构造函数调用之前被指定，构造函数的执行顺序大致如下图所示：



3.3 静态字段、属性和实例字段、属性

下面就直接总结下它们之间的区别：

- 静态成员包括静态属性和静态字段，静态字段一般实现为private,静态属性一般实现为public,从而来体现类的封装性
- 静态成员和类相关联，不依赖于对象而存在，只能由类来访问；实例成员与具体类相关联，只能由对象实例访问
- 静态成员不管创建多少实例对象，都在内存中只有一份，实例成员每创建一个实例对象，都会在内存中分配一块内存区域。

3.4 静态方法与实例方法

类似于静态字段和属性，静态方法共享代码段，同样以static关键字来标识静态方法，对于他们之间的区别总结为：

- 静态方法只能访问静态成员和方法，但是可以间接通过创建实例对象来访问实例字段、属性和方法；实例方法既可以访问实例成员也可以访问静态成员
- 静态方法由类方法‘实例方法由对象访问
- 静态方法不能引用this关键字，而实例方法可以
- 静态方法不能被标识为virtual、abstract或override,静态方法可以被派生访问，

但是不能被派生类重写
- Main方法为静态的，所以Main方法不能直接访问类中的实例字段、属性和方法，否则编译器会报错
- 静态方法一般用于作为通用的工具类来实现
- 在性能上，静态方法和实例方法的差别不大。因为，它们都是在JIT加载类的时候分配内存的，不同的是静态方法是以类为引用，而实例方法是以对象为引用，创建实例时，不会再为静态方法分配内存，所有实例对象共用一个类的方法代码，所以，静态方法和实例方法的调用，区别仅在于静态方法可以直接调用，而实例方法需要当前对象指针指向该方法，在性能上差不并不大。

四、小结

到这里，本文章的内容就介绍完了，通过对静态特征和非静态特征的由来来揭开一些都是源于生活的观点，然后再详细分析了静态特征与非静态特征在C#语言中的区别，希望这些总结可以帮助大家在复习基础知识的时候可以有用。同时也是自己的一个复习笔记的。

[C# 基础知识系列]C#中易混淆的知识点

一、引言

今天在论坛中看到一位朋友提出这样的一个问题，问题大致（问题的链接为：http://social.msdn.microsoft.com/Forums/zh-CN/52e6c11f-ad28-4633-a434-fc4d09f4d23d ）是这样的：

```
   static void Main(string[] args)
         {
             object m1 =1 ;
             object m2 = 1;

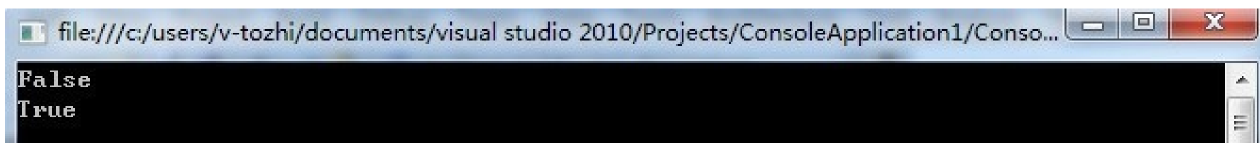
             Console.WriteLine(m1==m2);

             Console.WriteLine(m1.Equals(m2));
             Console.Read();
 }
```

大家先不要去Visual Studio中运行这段代码，先猜猜此段代码的运行结果是怎样的，如果你猜测的结果和运行出来的结果完全是一致并且你也知道原因的话，那这篇文章下面的内容就没必要看下去了，如果你对运行出来的结果表示不理解的话，那请继续看下面内容的分析，相信看完你绝对可以解除你的疑惑。

二、==与Equals的区别

上面问题的运行结果为：



对于结果为什么是这样的呢？这主要涉及到==与Equals方法的区别的，再讲两者的区别前，大家首先要明确——C#中有两种不同的相等：引用相等和值相等。值相等意味着两个对象保护相同的值，例如，两个值为1的整数就具有值相等性；引用相等意味着要比较的不是两个对象，而是两个对象的引用，且两者引用的是同一个对象。若要检查引用相等性，应使用 ReferenceEquals.aspx)。若要检查值相等性，请使用 Equals.aspx)（详细内容可以参考：http://msdn.microsoft.com/zh-cn/library/ms173147(v=vs.90).aspx.aspx)）。下面就看看它们直接的区别：

- ==比较的是栈内的内容，对于值类型而言，"=="比较的就是两个对象的值，除字符串（字符串类型是一个特殊情况）以外的引用类型比较的就是两个引用类型在栈内的地址
- Equals方法是定义在Object中的虚方法，用来比较两者引用对象的值是否相

等，.NET中类型就都可以重写Equals方法，例如，在.NET中string类型就重写了Equals方法，用于比较两个字符串的值是否相等，而不是字符串引用是否相等。

有了上面的理论基础，下面就具体分析上面程序为什么会是那样的结果：

1. 首先m1,m2都是引用类型，当执行m1==m2操作时，比较的是m1与m2在栈内地址的值是否相等，即比较的是引用，因为m1和m2指向的是托管堆中1是不同的地址（这点大家可以通过在debug状态下内存窗口中查看），所以得到的结果就自然是false
2. 对于m1.Equals(m2)比较的是m1与m2引用的值是否相等，因为它们都是引用托管堆中1，它们地址不等，但是值是相等的，都是1，所以返回为true。

下面用一道题目测试大家的掌握程度(也是为了进一步加深理解)

```
          static void Main(string[] args)
        {
            string str1 = "ZhangSan";
            string str2 = "ZhangSan";
            string str3 = new string(new char[] { 'z', 'h' });
            string str4 = new string(new char[] { 'z', 'h'});
            Console.WriteLine("str1 == str2  " + (str1 == str2).ToS
            Console.WriteLine("str1 Equals str2  " + str1.Equals(st

            Console.WriteLine("str3 == str4 " + (str3 == str4).ToSt
            Console.WriteLine("str3 Equals str4 " + str3.Equals(str

            Console.Read();
        }
```

View Code

运行结果为：

```
file:///c:/users/v-tozhi/documents/visual studio 2010/Projects/Conso
str1 == str2  True
str1 Equals str2  True
str3 == str4 True
str3 Equals str4 True
```

三、typeof与GetType区别

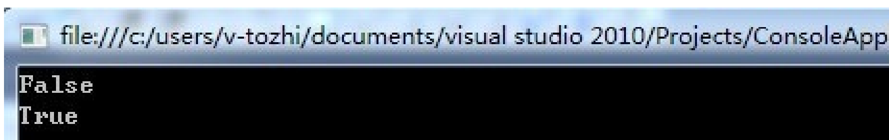
从上面那个问题中，我又联系到了typeof与GetType的区别，所以这里就一起总结下，首先我还是由一个程序来引出它们的区别：

```
            static void Main(string[] args)
        {
            object m1 = 1;
            object m2 = 1;
            // ValueType是引用类型，因为它是类，所以返回为false
            Console.WriteLine(typeof(ValueType).IsValueType);
            Console.WriteLine(m1.GetType().IsValueType);
            Console.Read();
        }
```

要想弄明白上面的运行结果，首先我们应该理解typeof与GetType的区别（之前我认为两个的都是一样的，这是一个误区），具体的区别为：

- typeof 是运算符，而 GetType() 是方法
- typeof 获得类型的**System.Type**对象，GetType()获得当前实例的Type，
- GetType()是基类System.Object的方法，只有建立了一个实例之后才能够被调用
- typeof的参数只能是int, string, class，自定义类型，不能为具体实例，否则编译器会报错

知道它们的区别之后，结果也就很容易得到了，上面程序的运行结果为：



四、小结

这篇文章主要是记录下自己在回答问题时所学到的内容，也希望对有同样疑惑的朋友有所帮助。

C#进阶系列

[C#进阶系列]专题一：深入解析深拷贝和浅拷贝

一、前言

这个星期参加了一个面试，面试中问到深浅拷贝的区别，然后我就简单了讲述了它们的之间的区别，然后面试官又继续问，如何实现一个深拷贝呢？当时只回答回答了一种方式，就是使用反射，然后面试官提示还可以通过反序列化和表达树的方式。然后又继续问，如果用反射来实现深拷贝的话，如何解决互相引用对象的问题呢？ 当时我给出的答案是说那就不用反射去实现呗，用反序列化实现呗，或者直接避免使两个对象互相引用呗。然后面试官说，如果一定用反射来写，你是怎么去解决这个问题呢？这时候我就愣住了。

这样也就有了这篇文章。今天就来深入解析下深浅拷贝的问题。

二、深拷贝 Vs 浅拷贝

首先，讲到深浅拷贝，自然就有一个问题来了？什么是深拷贝，什么又是浅拷贝呢？下面就具体介绍下它们的定义。

深拷贝：指的是拷贝一个对象时，不仅仅把对象的引用进行复制，还把该对象引用的值也一起拷贝。这样进行深拷贝后的拷贝对象就和源对象互相独立，其中任何一个对象的改动都不会对另外一个对象造成影响。举个例子，一个人叫张三，然后使用克隆技术以张三来克隆另外一个人叫李四，这样张三和李四就是相互独立的，不管张三缺胳膊还是李四少腿了都不会影响另外一个人。在.NET领域，值对象就是典型的例子，如int, Double以及结构体和枚举等。具体例子如下所示：

```
int source = 123;
// 值类型赋值内部执行深拷贝
int copy = source;
// 对拷贝对象进行赋值不会改变源对象的值
copy = 234;
// 同样对源对象赋值也不会改变拷贝对象的值
source = 345;
```

浅拷贝：指的是拷贝一个对象时，仅仅拷贝对象的引用进行拷贝，但是拷贝对象和源对象还是引用同一份实体。此时，其中一个对象的改变都会影响到另一个对象。例如，一个人一开始叫张三，后来改名字为张老三了，可是他们还是同一个人，不管张三缺胳膊还是张老三少腿，都反应在同一个人身上。在**.NET**中引用类型就是一个例子。如类类型。具体例子如下所示：

```
public class Person
    {
        public string Name { get; set; }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Person sourceP = new Person() { Name = "张三" };
            Person copyP = sourceP; // 浅拷贝
            copyP.Name = "张老三"; // 拷贝对象改变Name值
            // 结果都是"张老三",因为实现的是浅拷贝，一个对象的改变都会影响到
            Console.WriteLine("Person.Name: [SourceP: {0}] [CopyP:
            Console.Read();
        }
    }
```

三、深浅拷贝的几种实现方式

上面已经明白了深浅拷贝的定义，至于他们之间的区别也在定义中也有所体现。介绍完了它们的定义和区别之后，自然也就有了如何去实现它们呢？

对于，浅拷贝的实现方式很简单，.NET自身也提供了实现。我们知道，所有对象的父对象都是System.Object对象，这个父对象中有一个MemberwiseClone方法，该方法就可以用来实现浅拷贝，下面具体看看浅拷贝的实现方式，具体演示代码如下所示：

```
// 继承ICloneable接口，重新其Clone方法
    class ShallowCopyDemoClass : ICloneable
    {
        public int intValue = 1;
        public string strValue = "1";
        public PersonEnum pEnum = PersonEnum.EnumA;
        public PersonStruct pStruct = new PersonStruct() {  StructV
        public Person pClass = new Person("1");
        public int[] pIntArray = new int[] { 1 };
        public string[] pStringArray = new string[] { "1" };

        #region ICloneable成员
        public object Clone()
        {
            return this.MemberwiseClone();
        }

        #endregion

    }

    class Person
    {
        public string Name;
        public Person(string name)
        {
            Name = name;
        }
    }

    public enum PersonEnum
    {
        EnumA = 0,
        EnumB = 1
    }

    public struct PersonStruct
    {
        public int StructValue;
    }
```

上面类中重写了IConeable接口的Clone方法，其实现直接调用了Object的MemberwiseClone方法来完成浅拷贝，如果想实现深拷贝，也可以在Clone方法中实现深拷贝的逻辑。接下来就是对上面定义的类进行浅拷贝测试了，看看是否是实现的浅拷贝，具体演示代码如下所示：

```
class Program
    {
        static void Main(string[] args)
        {
            ShallowCopyDemo();
            // List浅拷贝的演示
            ListShallowCopyDemo();
        }

        public static void ListShallowCopyDemo()
        {
            List<PersonA> personList = new List<PersonA>()
            {
                new PersonA() { Name="PersonA", Age= 10, ClassA= ne
                new PersonA() { Name="PersonA2", Age= 20, ClassA= n
            };
            // 下面2种方式实现的都是浅拷贝
            List<PersonA> personsCopy = new List<PersonA>(personLis
            PersonA[] personCopy2 = new PersonA[2];
            personList.CopyTo(personCopy2);

  // 由于实现的是浅拷贝，所以改变一个对象的值，其他2个对象的值都会发生改变，因
            personsCopy.First().ClassA.TestProperty = "AProperty3",
            WriteLog(string.Format("personCopy2.First().ClassA.Test
            WriteLog(string.Format("personList.First().ClassA.TestP
            WriteLog(string.Format("personsCopy.First().ClassA.Test
  Console.Read();
        }

        public static void ShallowCopyDemo()
        {
            ShallowCopyDemoClass DemoA = new ShallowCopyDemoClass()
            ShallowCopyDemoClass DemoB = DemoA.Clone() as ShallowCo
            DemoB.intValue = 2;
            WriteLog(string.Format("    int->[A:{0}] [B:{1}]", Demo
            DemoB.strValue = "2";
            WriteLog(string.Format("    string->[A:{0}] [B:{1}]", D
            DemoB.pEnum = PersonEnum.EnumB;
            WriteLog(string.Format("  Enum->[A: {0}] [B:{1}]", Demo
            DemoB.pStruct.StructValue = 2;
            WriteLog(string.Format("    struct->[A: {0}] [B: {1}]",
            DemoB.pIntArray[0] = 2;
            WriteLog(string.Format("   intArray->[A:{0}] [B:{1}]",
            DemoB.pStringArray[0] = "2";
            WriteLog(string.Format("stringArray->[A:{0}] [B:{1}]",
            DemoB.pClass.Name = "2";
            WriteLog(string.Format("      Class->[A:{0}] [B:{1}]",
  Console.WriteLine();
}
```

```
private static void WriteLog(string msg) { Console.WriteLine(msg);
```

上面代码的运行结果如下图所示：

```
file:///F:/Study/C#/博客园中例子/DeepCopy/DeepCopy/bin/Debug/DeepCopy.EXE
    int->[A:1] [B:2]
    string->[A:1] [B:2]
  Enum->[A: EnumA] [B:EnumB]
    struct->[A: 1] [B: 2]
   intArray->[A:2] [B:2]
stringArray->[A:2] [B:2]
      Class->[A:2] [B:2]

personCopy2.First().ClassA.TestProperty is AProperty3
personList.First().ClassA.TestProperty is AProperty3
personsCopy.First().ClassA.TestProperty is AProperty3
```

从上面运行结果可以看出，.NET中值类型默认是深拷贝的，而对于引用类型，默认实现的是浅拷贝。所以对于类中引用类型的属性改变时，其另一个对象也会发生改变。

上面已经介绍了浅拷贝的实现方式，那深拷贝要如何实现呢？在前言部分已经介绍了，实现深拷贝的方式有：反射、反序列化和表达式树。在这里，我只介绍反射和反序列化的方式，对于表达式树的方式在网上也没有找到，当时面试官说是可以的，如果大家找到了表达式树的实现方式，麻烦还请留言告知下。下面我们首先来看看反射的实现方式吧：

```
// 利用反射实现深拷贝
        public static T DeepCopyWithReflection<T>(T obj)
        {
            Type type = obj.GetType();

            // 如果是字符串或值类型则直接返回
            if (obj is string || type.IsValueType) return obj;

            if (type.IsArray)
            {
                Type elementType = Type.GetType(type.FullName.Repla
                var array = obj as Array;
                Array copied = Array.CreateInstance(elementType, ar
                for (int i = 0; i < array.Length; i++)
                {
                    copied.SetValue(DeepCopyWithReflection(array.Ge
                }

                return (T)Convert.ChangeType(copied, obj.GetType()
            }

            object retval = Activator.CreateInstance(obj.GetType()

            PropertyInfo[] properties = obj.GetType().GetProperties
                BindingFlags.Public | BindingFlags.NonPublic
                | BindingFlags.Instance | BindingFlags.Static);
            foreach (var property in properties)
            {
                var propertyValue = property.GetValue(obj, null);
                if (propertyValue == null)
                    continue;
                property.SetValue(retval, DeepCopyWithReflection(pr
            }

            return (T)retval;
        }
```

反序列化的实现方式，反序列化的方式也可以细分为3种，具体的实现如下所示：

```
  // 利用XML序列化和反序列化实现
        public static T DeepCopyWithXmlSerializer<T>(T obj)
        {
            object retval;
            using (MemoryStream ms = new MemoryStream())
            {
                XmlSerializer xml = new XmlSerializer(typeof(T));
                xml.Serialize(ms, obj);
                ms.Seek(0, SeekOrigin.Begin);
                retval = xml.Deserialize(ms);
                ms.Close();
```

```
            }

            return (T)retval;
        }

        // 利用二进制序列化和反序列实现
        public static T DeepCopyWithBinarySerialize<T>(T obj)
        {
            object retval;
            using (MemoryStream ms = new MemoryStream())
            {
                BinaryFormatter bf = new BinaryFormatter();
                // 序列化成流
                bf.Serialize(ms, obj);
                ms.Seek(0, SeekOrigin.Begin);
                // 反序列化成对象
                retval = bf.Deserialize(ms);
                ms.Close();
            }

            return (T)retval;
        }

        // 利用DataContractSerializer序列化和反序列化实现
        public static T DeepCopy<T>(T obj)
        {
            object retval;
            using (MemoryStream ms = new MemoryStream())
            {
                DataContractSerializer ser = new DataContractSerial
                ser.WriteObject(ms, obj);
                ms.Seek(0, SeekOrigin.Begin);
                retval = ser.ReadObject(ms);
                ms.Close();
            }
            return (T)retval;
        }

        // 表达式树实现
        // ....
```

四、使用反射进行深拷贝如何解决相互引用的问题

上面反射的实现方式，对于相互引用的对象会出现StackOverflower的错误，由于对象的相互引用，会导致方法循环调用。下面就是一个相互引用对象的例子：

```
[Serializable]
    public class DeepCopyDemoClass
    {
        public string Name {get;set;}
        public int[] pIntArray { get; set; }
        public Address Address { get; set; }
        public DemoEnum DemoEnum { get; set; }

        // DeepCopyDemoClass中引用了TestB对象，TestB类又引用了DeepCopyI
        public TestB TestB {get;set;}

        public override string ToString()
        {
            return "DeepCopyDemoClass";
        }
    }

    [Serializable]
    public class TestB
    {
        public string Property1 { get; set; }

        public DeepCopyDemoClass DeepCopyClass { get; set; }

        public override string ToString()
        {
            return "TestB Class";
        }
    }

    [Serializable]
    public struct Address
    {
        public string City { get; set; }
    }

    public enum DemoEnum
    {
        EnumA = 0,
        EnumB = 1
    }
```

在面试过程中，针对这个问题的解决方式我回答的是不知道，回来之后思考了之后，也就有了点思路。首先想到的是：能不能用一个字典来记录每个对象被反射的次数，仔细想想可行，于是开始实现，初步修复后的反射实现如下所示：

```
  public class DeepCopyHelper
    {
        // 用一个字典来存放每个对象的反射次数来避免反射代码的循环递归
```

```
        static Dictionary<Type, int> typereflectionCountDic = new D

   public static T DeepCopyWithReflection_Second<T>(T obj)
        {
            Type type = obj.GetType();

            // 如果是字符串或值类型则直接返回
            if (obj is string || type.IsValueType) return obj;

            if (type.IsArray)
            {
                Type elementType = Type.GetType(type.FullName.Repla
                var array = obj as Array;
                Array copied = Array.CreateInstance(elementType, ar
                for (int i = 0; i < array.Length; i++)
                {
                    copied.SetValue(DeepCopyWithReflection_Second(a
                }

                return (T)Convert.ChangeType(copied, obj.GetType()
            }

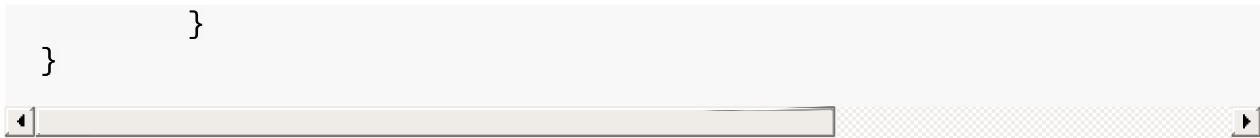
            // 对于类类型开始记录对象反射的次数
            int reflectionCount = Add(typereflectionCountDic, obj.G
            if (reflectionCount > 1)
                return obj; // 这里有错误

            object retval = Activator.CreateInstance(obj.GetType()

            PropertyInfo[] properties = obj.GetType().GetProperties
                BindingFlags.Public | BindingFlags.NonPublic
                | BindingFlags.Instance | BindingFlags.Static);
            foreach (var property in properties)
            {
                var propertyValue = property.GetValue(obj, null);
                if (propertyValue == null)
                    continue;
                property.SetValue(retval, DeepCopyWithReflection_Se
            }

            return (T)retval;
        }
        private static int Add(Dictionary<Type, int> dict, Type key
        {
            if (key.Equals(typeof(String)) || key.IsValueType) retu
            if (!dict.ContainsKey(key))
            {
                dict.Add(key, 1);
                return dict[key];
            }

            dict[key] += 1;
            return dict[key];
```



```
        }
}
```

下面用代码来测试下上面的代码是否已经解决了循环递归的问题，具体的测试代码如下所示：

```
class Program
    {
        static void Main(string[] args)
        {
            //ShallowCopyDemo();
            //ListShallowCopyDemo();
            DeepCopyDemo();
            DeepCopyDemo2();
        }
        private static void WriteLog(string msg)
        {
            Console.WriteLine(msg);
        }

        public static void DeepCopyDemo()
        {
            DeepCopyDemoClass deepCopyClassA = new DeepCopyDemoClas
            deepCopyClassA.Name = "DeepCopyClassDemo";
            deepCopyClassA.pIntArray = new int[] { 1 };
            deepCopyClassA.DemoEnum = DemoEnum.EnumA;
            deepCopyClassA.Address = new Address() { City = "Shangh

            deepCopyClassA.TestB = new TestB() { Property1 = "TestP

            // 使用反序列化来实现深拷贝
            DeepCopyDemoClass deepCopyClassB = DeepCopyHelper.DeepC
            deepCopyClassB.Name = "DeepCopyClassDemoB";
            WriteLog(string.Format("    Name->[A:{0}] [B:{1}]", dee
            deepCopyClassB.pIntArray[0] = 2;
            WriteLog(string.Format("    intArray->[A:{0}] [B:{1}]",
            deepCopyClassB.Address = new Address() { City = "Beijin
            WriteLog(string.Format("    Addressstruct->[A: {0}] [B
            deepCopyClassB.DemoEnum = DemoEnum.EnumB;
            WriteLog(string.Format("    DemoEnum->[A: {0}] [B: {1}]
            deepCopyClassB.TestB.Property1 = "TestPropertyB";
            WriteLog(string.Format("    Property1->[A:{0}] [B:{1}]"
            WriteLog(string.Format("    TestB.DeepCopyClass.Name->[
            Console.WriteLine();
        }

        public static void DeepCopyDemo2()
        {
            DeepCopyDemoClass deepCopyClassA = new DeepCopyDemoClas
            deepCopyClassA.Name = "DeepCopyClassDemo";
```

```
            deepCopyClassA.pIntArray = new int[] { 1, 2 };
            deepCopyClassA.DemoEnum = DemoEnum.EnumA;
            deepCopyClassA.Address = new Address() { City = "Shangh

            deepCopyClassA.TestB = new TestB() { Property1 = "TestP

            // 使用反射来完成深拷贝
            DeepCopyDemoClass deepCopyClassB = DeepCopyHelper.DeepC

            //DeepCopyDemoClass deepCopyClassB = DeepCopyHelper.Dee
            deepCopyClassB.Name = "DeepCopyClassDemoB";
            WriteLog(string.Format("    Name->[A:{0}] [B:{1}]", dee
            deepCopyClassB.pIntArray[0] = 2;
            WriteLog(string.Format("    intArray->[A:{0}] [B:{1}]",
            deepCopyClassB.Address = new Address() { City = "Beijin
            WriteLog(string.Format("    Addressstruct->[A: {0}] [B
            deepCopyClassB.DemoEnum = DemoEnum.EnumB;
            WriteLog(string.Format("    DemoEnum->[A: {0}] [B: {1}]
            deepCopyClassB.TestB.Property1 = "TestPropertyB";
            WriteLog(string.Format("    Property1->[A:{0}] [B:{1}]"
            WriteLog(string.Format("    TestB.DeepCopyClass.Name->[
            Console.ReadKey();
        }
    }
```

此时的运行结果如下图所示：

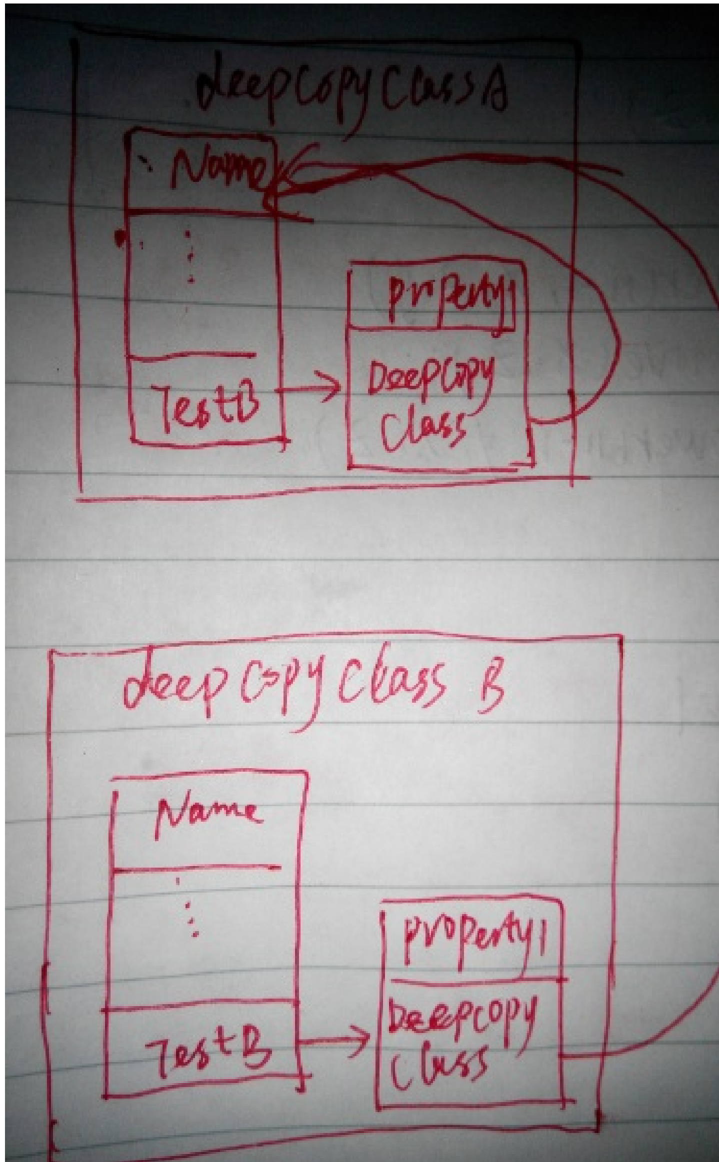
```
file:///F:/Study/C#/博客园中例子/DeepCopy/DeepCopy/bin/Debug/DeepCopy.EXE
    Name->[A:DeepCopyClassDemo] [B:DeepCopyClassDemoB]
    intArray->[A:1] [B:2]
    Addressstruct->[A: Shanghai] [B: Beijing]
    DemoEnum->[A: EnumA] [B: EnumB]
    Property1->[A:TestProperty] [B:TestPropertyB]
    TestB.DeepCopyClass.Name->[A:DeepCopyClassDemo] [B:DeepCopyClassDemoB]

    Name->[A:DeepCopyClassDemo] [B:DeepCopyClassDemoB]
    intArray->[A:1] [B:2]
    Addressstruct->[A: Shanghai] [B: Beijing]
    DemoEnum->[A: EnumA] [B: EnumB]
    Property1->[A:TestProperty] [B:TestPropertyB]
    TestB.DeepCopyClass.Name->[A:DeepCopyClassDemo] [B:DeepCopyClassDemo]
```

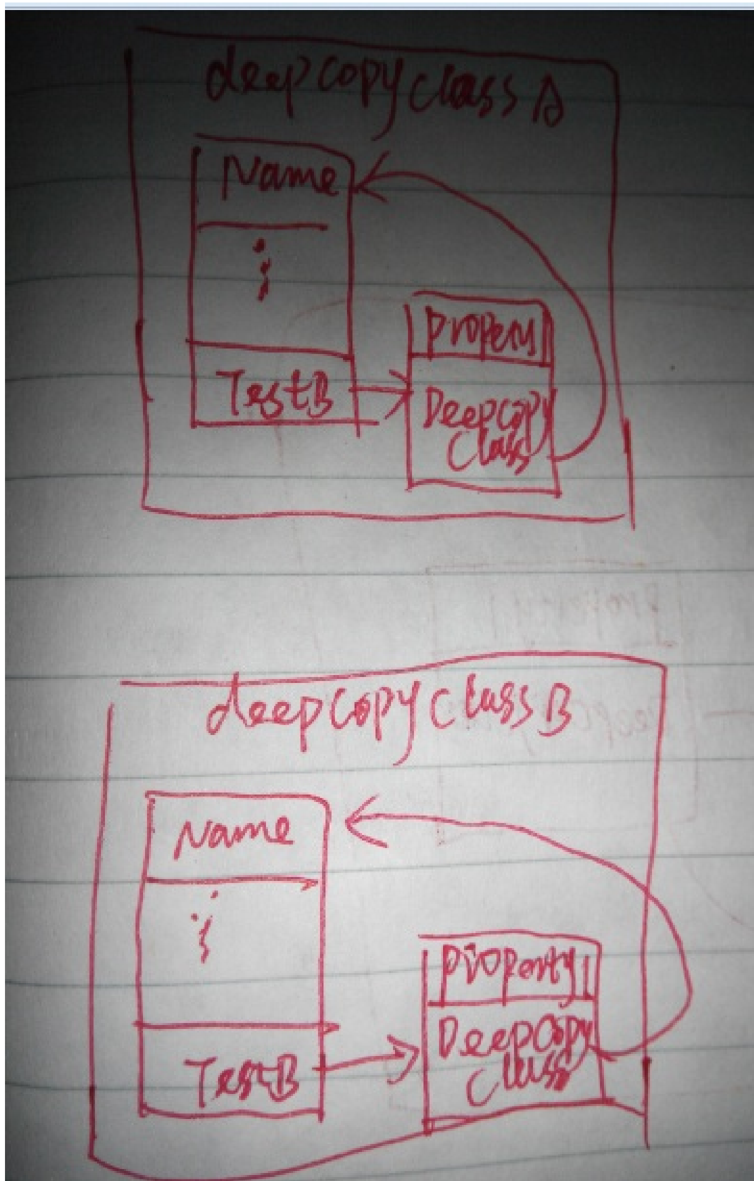
刚开始看到这样的运行结果，开心地以为已经解决了循环递归的问题了，因为此时结果成功运行出来了，没有了StackOverflower的错误了。但是仔细一看，反序列化和反射完成的深拷贝的运行结果不一样，如上图中红色圈出来的部分。显然，反序列化的结果是没有错误的，显然目前实现的反射代码还是有问题的。接下来就是思考了。为什么上面反射的代码不正确呢？

仔细分析DeepCopyWithReflection_Second中的代码，发现下面代码红色部分是错误的：

对DeepCopyWithReflection_Second方法仔细分析，在对TestB进行反射时，当反射到DeepCopyClass属性时，此时会递归调用DeepCopyWithReflection_Second方法，此时在typereflectionCountDic发现DeepCopyDemoClass已经被反射了，则直接返回，这样分析好像没什么错误，但是此时返回的是deepCopyClassA对象，但是我们需要返回的是deepCopyClassB对象，即此时deepCopyClassB对象的内存结构如下图所示：



而我们其实需要deepCopyClassB对象的内存结构如下图所示：



既然找到了DeepCopyWithReflection_Second的错误原因，那我们就要解决了。上面说我们返回的应该是deepCopyClassB对象，而我们怎么得到创建的deepCopyClassB对象呢？这里我就想能不能用一个变量来保存一开始通过CreateInstance方法创建的deepCopyClassB对象呢？验证想法最好的办法就是代码了，这样我就按照这个思路对DeepCopyWithReflection_Second又进行一次改进，最终的代码如下所示：

```
public static T DeepCopyWithReflection_Third<T>(T obj)
        {
            Type type = obj.GetType();

            // 如果是字符串或值类型则直接返回
            if (obj is string || type.IsValueType) return obj;

            if (type.IsArray)
            {
                Type elementType = Type.GetType(type.FullName.Repla
                var array = obj as Array;
                Array copied = Array.CreateInstance(elementType, ar
                for (int i = 0; i < array.Length; i++)
                {
                    copied.SetValue(DeepCopyWithReflection_Second(a
                }

                return (T)Convert.ChangeType(copied, obj.GetType()
            }

            int reflectionCount = Add(typereflectionCountDic, obj.G
            if (reflectionCount > 1 && obj.GetType() == typeof(Deep
                return (T)DeepCopyDemoClasstypeRef; // 返回deepCopy

            object retval = Activator.CreateInstance(obj.GetType()

            if(retval.GetType() == typeof(DeepCopyDemoClass))
                DeepCopyDemoClasstypeRef = retval; // 保存一开始创建的

            PropertyInfo[] properties = obj.GetType().GetProperties
                BindingFlags.Public | BindingFlags.NonPublic
                | BindingFlags.Instance | BindingFlags.Static);
            foreach (var property in properties)
            {
                var propertyValue = property.GetValue(obj, null);
                if (propertyValue == null)
                    continue;
                property.SetValue(retval, DeepCopyWithReflection_Th
            }

            return (T)retval;
        }
```

下面我用DeepCopyWithReflection_Third方法来测试下，具体的测试代码如下所示：

```
class Program
    {
        static void Main(string[] args)
        {
```

```
            //ShallowCopyDemo();
            //ListShallowCopyDemo();
            DeepCopyDemo();
            DeepCopyDemo2();
        }
         private static void WriteLog(string msg)
        {
            Console.WriteLine(msg);
        }

        public static void DeepCopyDemo()
        {
            DeepCopyDemoClass deepCopyClassA = new DeepCopyDemoClas
            deepCopyClassA.Name = "DeepCopyClassDemo";
            deepCopyClassA.pIntArray = new int[] { 1 };
            deepCopyClassA.DemoEnum = DemoEnum.EnumA;
            deepCopyClassA.Address = new Address() { City = "Shangh

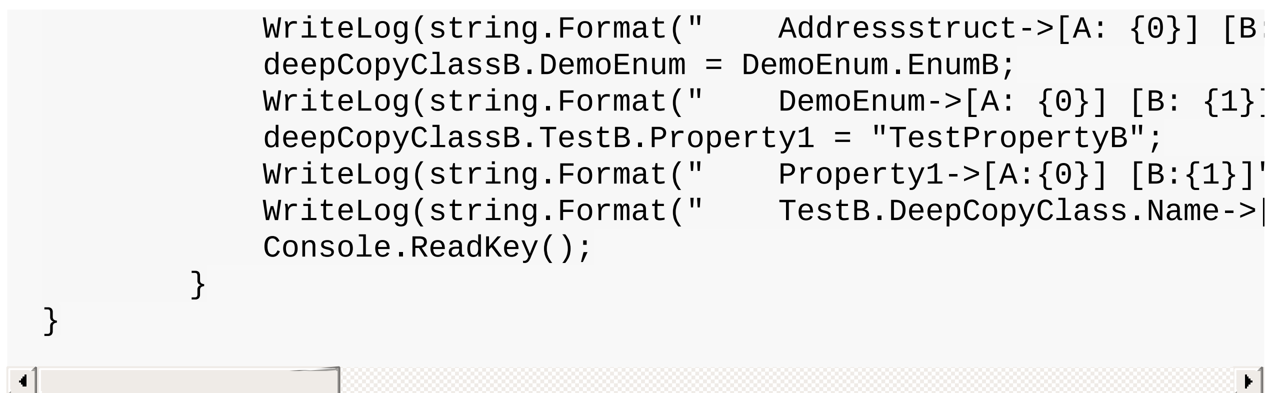
            deepCopyClassA.TestB = new TestB() { Property1 = "TestP

            // 使用反序列化来实现深拷贝
            DeepCopyDemoClass deepCopyClassB = DeepCopyHelper.DeepC
            deepCopyClassB.Name = "DeepCopyClassDemoB";
            WriteLog(string.Format("    Name->[A:{0}] [B:{1}]", dee
            deepCopyClassB.pIntArray[0] = 2;
            WriteLog(string.Format("    intArray->[A:{0}] [B:{1}]",
            deepCopyClassB.Address = new Address() { City = "Beijin
            WriteLog(string.Format("    Addressstruct->[A: {0}] [B:
            deepCopyClassB.DemoEnum = DemoEnum.EnumB;
            WriteLog(string.Format("    DemoEnum->[A: {0}] [B: {1}]
            deepCopyClassB.TestB.Property1 = "TestPropertyB";
            WriteLog(string.Format("    Property1->[A:{0}] [B:{1}]"
            WriteLog(string.Format("    TestB.DeepCopyClass.Name->[
            Console.WriteLine();
        }

        public static void DeepCopyDemo2()
        {
            DeepCopyDemoClass deepCopyClassA = new DeepCopyDemoClas
            deepCopyClassA.Name = "DeepCopyClassDemo";
            deepCopyClassA.pIntArray = new int[] { 1, 2 };
            deepCopyClassA.DemoEnum = DemoEnum.EnumA;
            deepCopyClassA.Address = new Address() { City = "Shangh

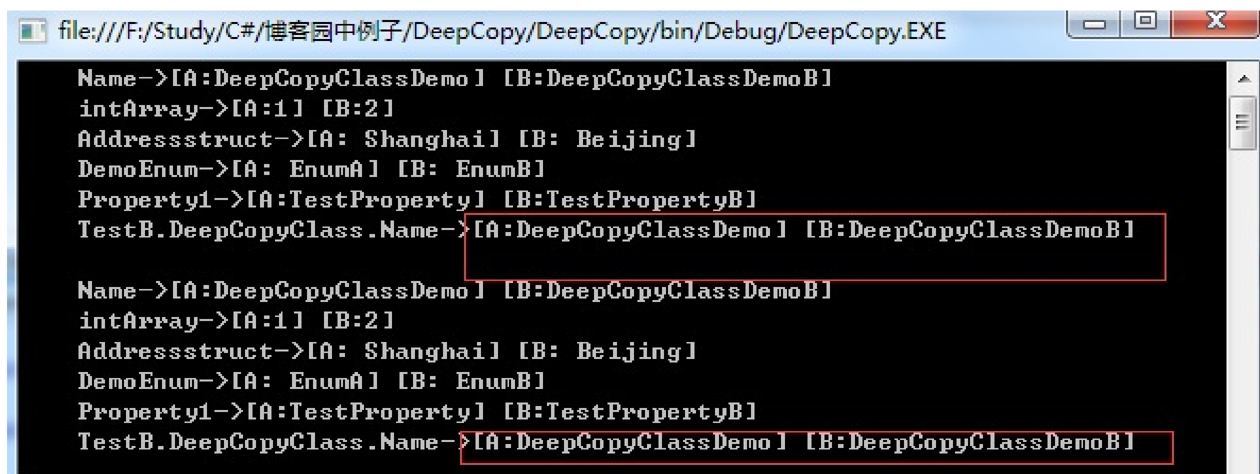
            deepCopyClassA.TestB = new TestB() { Property1 = "TestP

            **// 使用反射来完成深拷贝
            DeepCopyDemoClass deepCopyClassB = DeepCopyHelper.DeepC
            deepCopyClassB.Name = "DeepCopyClassDemoB";
            WriteLog(string.Format("    Name->[A:{0}] [B:{1}]", dee
            deepCopyClassB.pIntArray[0] = 2;
            WriteLog(string.Format("    intArray->[A:{0}] [B:{1}]",
            deepCopyClassB.Address = new Address() { City = "Beijin
```



```
                WriteLog(string.Format("    Addressstruct->[A: {0}] [B
                deepCopyClassB.DemoEnum = DemoEnum.EnumB;
                WriteLog(string.Format("    DemoEnum->[A: {0}] [B: {1}]
                deepCopyClassB.TestB.Property1 = "TestPropertyB";
                WriteLog(string.Format("    Property1->[A:{0}] [B:{1}]"
                WriteLog(string.Format("    TestB.DeepCopyClass.Name->
                Console.ReadKey();
            }
    }
```

此时的运行结果如下图示所示：



从上面的测试结果可以看出，此时深拷贝的反射实现方法基本上没什么问题了。这个方法也同时解决了相互引用对象的循环递归问题。

五、总结

到这里，该文章的内容就结束。这里主要记录下自己在一次面试过程中遇到问题的一次总结，从中可以看出，反射进行深拷贝会有很多其他的问题，所以平时还是建议大家使用序列化的形式来进行深拷贝。

最后附上本文所有源码下载：DeepCopy.zip

[C#进阶系列]专题二：你知道Dictionary查找速度为什么快吗？

一、前言

在之前有一次面试中，被问到你了解Dictionary的内部实现机制吗？当时只是简单的了问答了：Dictionary的内部结构是哈希表，从而可以快速进行查找。但是对于更深一步了解就不清楚了。所以面试回来之后，就打算好好研究下Dictionary的源码。所以也就有了这篇文章。

二、Dictionary源码剖析

大家都知道，现在微软已经开源了.NET Framework的源码了，在线源码查看地址为：http://referencesource.microsoft.com/。通过查找可以找到.NET Framework类的源码。下面我们就一起来看下Dictionary源码。

2.1 添加元素

首先我们来查看下Dictionary.Add方法的实现。为了让大家更好地实现，下面抽取了Dictionary源码核心部分来进行分析，详细的分析代码如下所示：

```
// buckets是哈希表，用来存放Key的Hash值
        // entries用来存放元素列表
        // count是元素数量
        private void Insert(TKey key, TValue value, bool add)
        {
            if (key == null)
            {
                throw new ArgumentNullException(key.ToString());
            }
            // 首先分配buckets和entries的空间
            if (buckets == null) Initialize(0);
            int hashCode = comparer.GetHashCode(key) & 0x7FFFFFFF;
            int targetBucket = hashCode % buckets.Length; // 对哈希

#if FEATURE_RANDOMIZED_STRING_HASHING
            int collisionCount = 0;
#endif
            // 处理冲突的处理逻辑
            for (int i = buckets[targetBucket]; i >= 0; i = entries
            {
                if (entries[i].hashCode == hashCode && comparer.Equ
                {
```

```
                    if (add)
                    {
                        throw new ArgumentNullException();
                    }
                    entries[i].value = value;
                    version++;
                    return;
                }

#if FEATURE_RANDOMIZED_STRING_HASHING
                collisionCount++;
#endif
            }

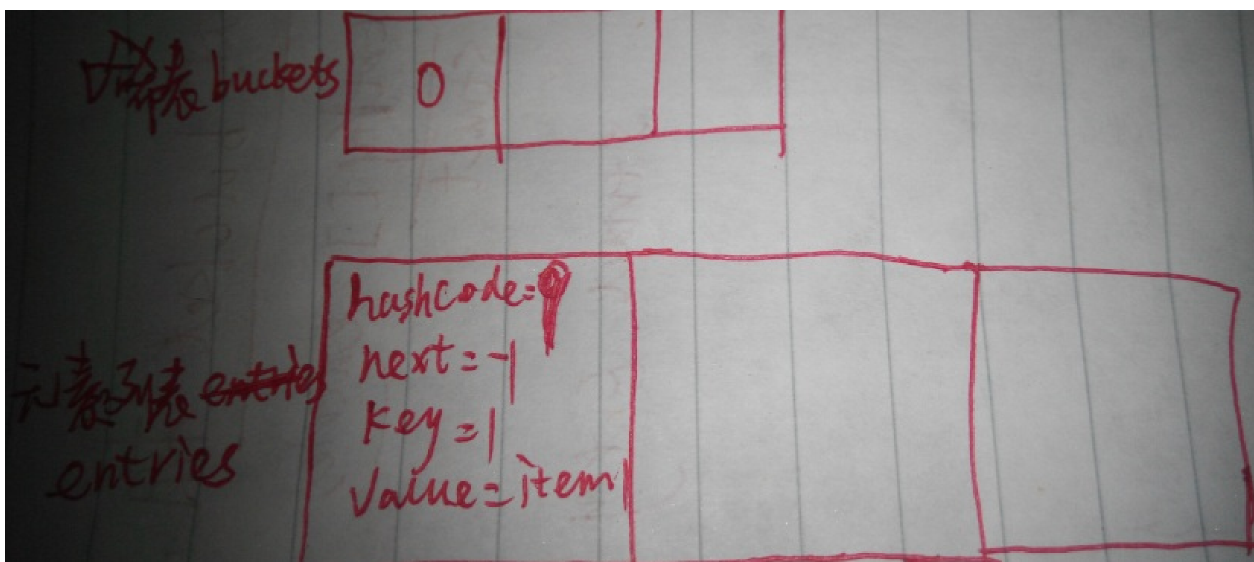
            int index; // index记录了元素在元素列表中的位置
            if (freeCount > 0)
            {
                index = freeList;
                freeList = entries[index].next;
                freeCount--;
            }
            else
            {
                // 如果哈希表存放哈希值已满，则重新从primers数组中取出值来
                if (count == entries.Length)
                {
                    Resize();
                    targetBucket = hashCode % buckets.Length;
                }
                // 大小如果没满的逻辑
                index = count;
                count++;
            }

            // 对元素列表进行赋值
            entries[index].hashCode = hashCode;
            entries[index].next = buckets[targetBucket];
            entries[index].key = key;
            entries[index].value = value;
            // 对哈希表进行赋值
            buckets[targetBucket] = index;
            version++;

#if FEATURE_RANDOMIZED_STRING_HASHING
            if(collisionCount > HashHelpers.HashCollisionThreshold
            {
                comparer = (IEqualityComparer<TKey>) HashHelpers.Ge
                Resize(entries.Length, true);
            }
#endif
        }
```

下面以一个实际的添加例子来具体分析下上面的添加元素代码，从而更好地理解Add方法的实现原理。

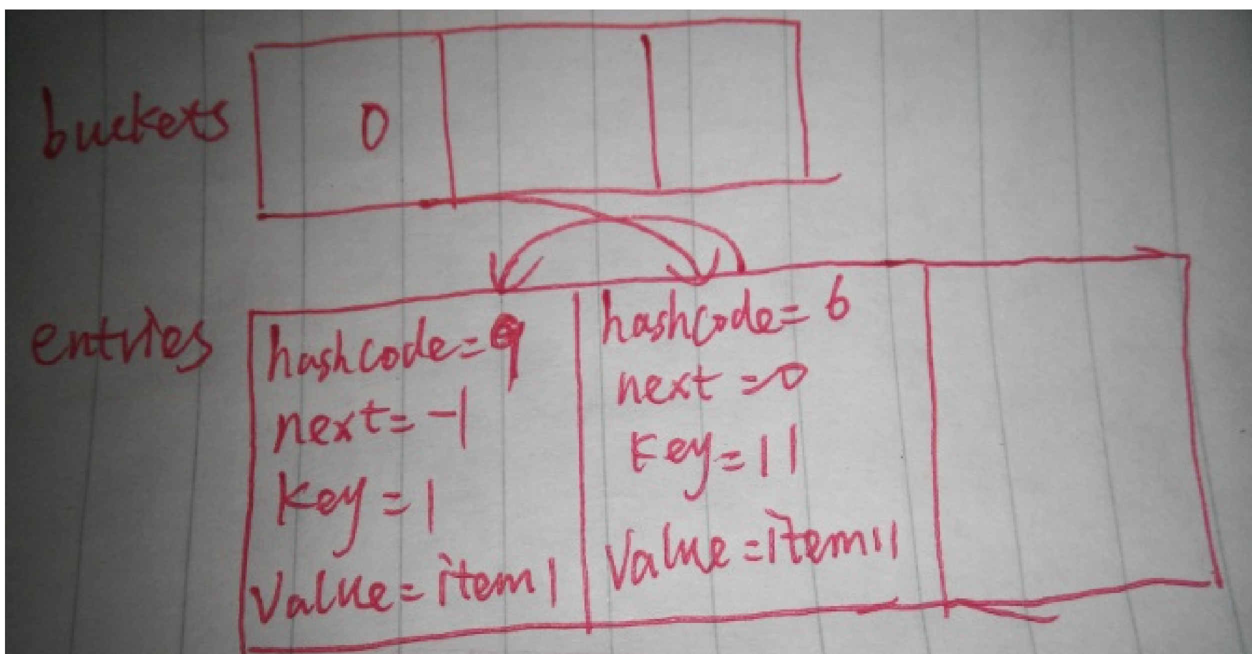
当添加第一个元素时，此时会分配哈希表buckets数组和entries数组的空间和初始大小为3，分配完成之后，会计算添加元素key值的哈希值，哈希值的计算由具体的哈希算法来实现的，假设1的哈希值为9的话，此时targetBucket = 9%buckets.Length(3)的值为0，index的值为0，则第一个元素存放在entries列表中的第一个位置，最后对哈希表进行赋值，此时赋值的位置为第0个位置，其值为index的值，所以为0，插入第一个元素后Dictionary的内部结构如下所示：



后面添加元素的过程依次类推。其原理就是，buckets记录了元素的在元素列表的存储位置，也就相当于一个映射列表。在查找的时候，就可以通过key值的哈希值来与buckets数组长度求余来获得元素在元素列表中的索引，这样就可以快速定位元素的位置，从而获得元素的key对应的Value值。如上面的例子中，如果想找到key值为1对应的Value值时，此时计算1的哈希值为9，然后对buckets数组长度求余，此时获得的值正是0，这样就可以直接从entries[0].Value的方式来获取对应的Value的值，这也就是Dictionary能完成快速查找的实现原理。后面会通过Dictionary内部的查找源码来证实上面分析的过程。

2.2 解决冲突

在添加元素过程中，有一个很重要的问题，如果产生冲突怎么办？即如果我后面需要插入的一个元素(假设这个值为11吧)的key值的哈希值也为6，此时targetBucket的值也是为0，但元素列表中0的位置已经存放了元素了，这样就出现了冲突，那Dictionary是怎样处理这个冲突的呢？处理冲突的方法有很多种，Dictionary处理的方式是链接法。Dictionary会把发生冲突的元素链接之前元素的后面，通过next属性来指定冲突关系。此时Dictionary内部结构如下图所示：



三、Dictionary如何实现快速查找呢？

针对于Dictionary实现快速查找的原因，在上面我们已经做了一个推断了，下面通过Dictionary内部的代码实现来验证下，具体的查找代码如下所示：

```
public TValue this[TKey key]
        {
            get
            {
                int i = FindEntry(key);
                // 通过元素所在存在的位置直接获取其对应的Value
                if (i >= 0) return entries[i].value;
                throw new KeyNotFoundException();
                return default(TValue);
            }
            set
            {
                Insert(key, value, false);
            }
        }

        private int FindEntry(TKey key)
        {
            if (key == null)
            {
                throw new ArgumentNullException();
            }

            if (buckets != null)
            {
                // 获得Key值对应的哈希值
                int hashCode = comparer.GetHashCode(key) & 0x7FFFF
                // 查找元素在元素列表中的位置，如果没有冲突的情况下，此时查
                for (int i = buckets[hashCode % buckets.Length]; i
                {
                    if (entries[i].hashCode == hashCode && compare
                }
            }
            return -1;
        }
```

通过代码可以看出，我们之前的分析是完成正确的。从中可以明白：Dictionary之所以能实现快速查找元素，其内部使用哈希表来存储元素对应的位置，然后我们可以通过哈希值快速地从哈希表中定位元素所在的位置索引，从而快速获取到key对应的Value值。

四、总结

可以说，Dictionary的实现原理也是一种空间换时间的思路，多使用一个buckets的存储空间来存储元素的位置，从而来提升查找速度。

接下来，我们新开一个领域驱动设计系列，还请大家多多拍砖。

本文所有源码下载：DictonaryInDepth.zip

C# 开发技巧系列

[C# 开发技巧系列]使用C#操作Word和Excel程序

一、引言

在我们日常办公中，我们经常可能遇到一些重复性的工作的，比如，我们在写毕业设计的时候，有时候我们写的过程中不注意，当整篇毕业论文写完之后，发现在毕业论文中存在很多空白的段落，这是我们就需要人工重新审阅一遍论文，再手动删除一些空白行，由于毕业论文也不是一篇，有开题报告啊，文献翻译等等，这样就可能需要我们人工都去审阅一篇把一些空白行删除，这样既花时间，我们也看的累。然后还有一个例子就是——我们人事部门的MM们，一到月末的时候就需要给本月的寿星员工发送邮件来通知参加生日会，如果员工信息是在Excel中的话，这时候人事的MM就要手动地从中查找本月寿星的邮箱，然后用Outlook一个一个添加邮件地址来给本月寿星发送邮件的，为了让人事MM们不再那么累，所以就想能不能用程序自动化地完成这一系列的过程呢？答案是肯定的，下面就让我来实现上面的两个需求的，使我们（尤其是人事MM们）的办公更加Easy。

二、自动删除**Word**中的空白行和页

在引言部分，我们已经提出了这个需求的。记得当时在写毕业论文的时候，我也做过这些重复的事情，经常写完之后会再去审阅一遍毕业论文中的所有文档，然后手动把一些空白行删除掉，由于当时并不知道可以对Word来进行自动化编程，所以只能傻傻地做这样一些重复的事情。但是现在不一样了，自从接触了VSTO之后，才知道Office一系列产品都是提供了一些公开的API的，我们可以利用这些对象使我们自定义Office程序和使Office自动化地工作，下面就具体讲讲如何实现这个小的工具的。

首先，我们先明确下这个工具需要实现的功能——自动移除Word文档中的空白行。然后向大家解释下实现该工具的思路：

- 我们打开一个Word文档，该Word文档就是一个**Word.Document**对象
- Word文档中内容都是段落组成的，然而段落在Word对象模型中是**Word.Paragraph**对象
- 空白行或空白页也就是段落的内容为空，明白了这点，我们就可以在程序中对段落对象的文本进行判断，如果段落内容为空，我们就删除该段落，这样也就实现了移除空白行的功能了。

有了上面的思路之后，然后大家只需要了解Word中对象模型，然后通过对象模型找到段落对象，然后再判断它的文本是否为空，为空就删除段落，不为空就什么都不做。所以思路有了之后，就是要了解Word对象模型了，对于这部分内容，大家可以参考我博客的中的——创建Word解决方案。由于代码中都有注释，这里就直接看实现该工具的核心代码：

```
string[] wordPatharray = null;
        // 打开需要操作的Word文档
        private void btnOpen_Click(object sender, EventArgs e)
        {
            using (OpenFileDialog openFileDialog = new OpenFileDial
```

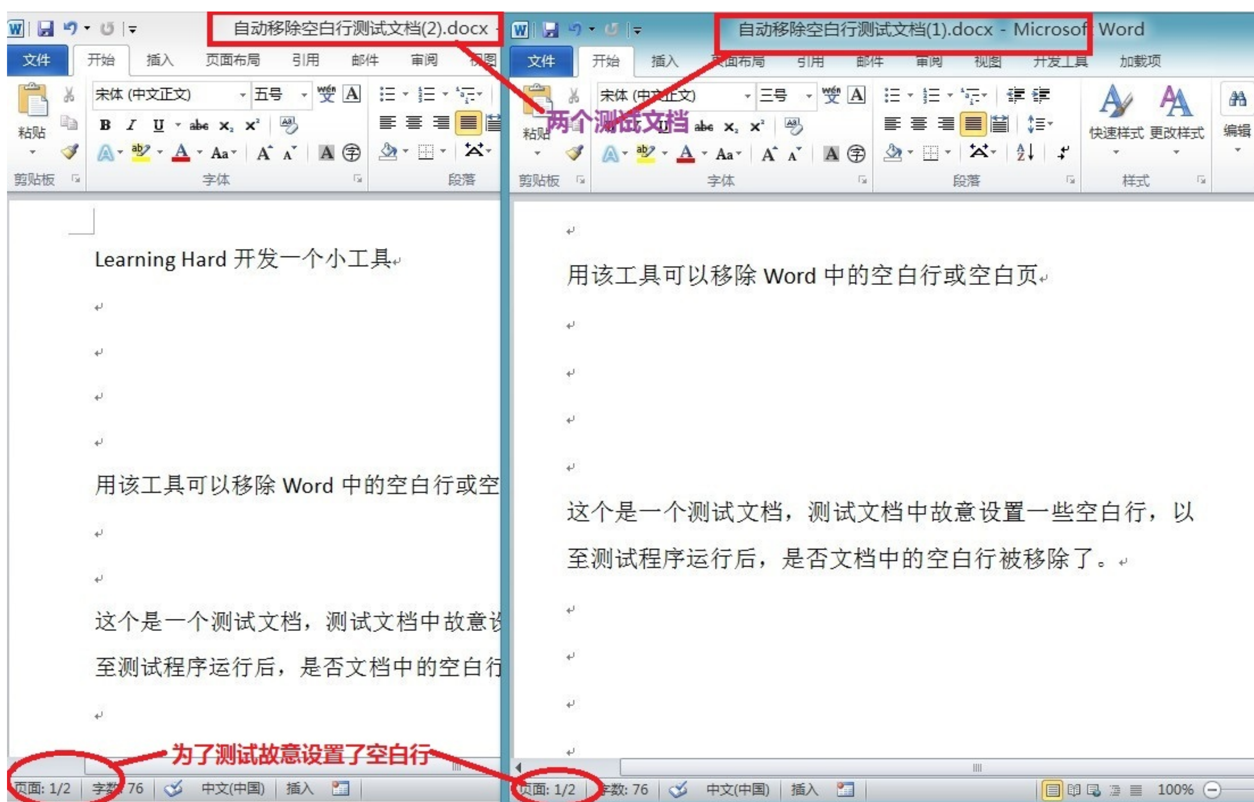
```
            {
                openFileDialog.Filter = "Word document(*.doc;*.docx
                // 设置允许选择多个文件，该属性默认为false的，即只允许选择-
                openFileDialog.Multiselect = true;
                if (openFileDialog.ShowDialog() == DialogResult.OK)
                {
                    txtWordPath.Text = openFileDialog.FileName;
                    // 获得所有选定文件的文件名
                    wordPatharray = openFileDialog.FileNames;
                }
            }
        }
        // 移除Word中的所有空页
        private void btnRemove_Click(object sender, EventArgs e)
        {
            Word.Application wordapp = null;
            Word.Document doc = null;
            try
            {
                // 启动Word应用程序并设置不可见
                wordapp = new Word.Application();
                // 如果不设置该属性，就可以看到Word程序的启动过程，这个和我
                wordapp.Visible = false;
                // 遍历每个文件名
                foreach (var wordpath in wordPatharray)
                {
                    doc = wordapp.Documents.Open(wordpath);
                    // 删除所有空白页面
                    Word.Paragraph paragraph;
                    Word.Paragraphs paragraphs = doc.Paragraphs;
                    for (int i = paragraphs.Count; i > 0; i--)
                    {
                        paragraph = paragraphs[i];

                        //  如果段落的文本为空的话，首先选择该段落，然后再
                        // 不为空什么都不做
                        if (paragraph.Range.Text.Trim() == string.E
                        {
                            paragraph.Range.Select();
                            wordapp.Selection.Delete();
                        }
                    }
                    if (doc != null)
                    {
                        // 先保存所有修改再关闭Word文档
                        doc.Save();
                        ((Word._Document)doc).Close();
                    }
                }

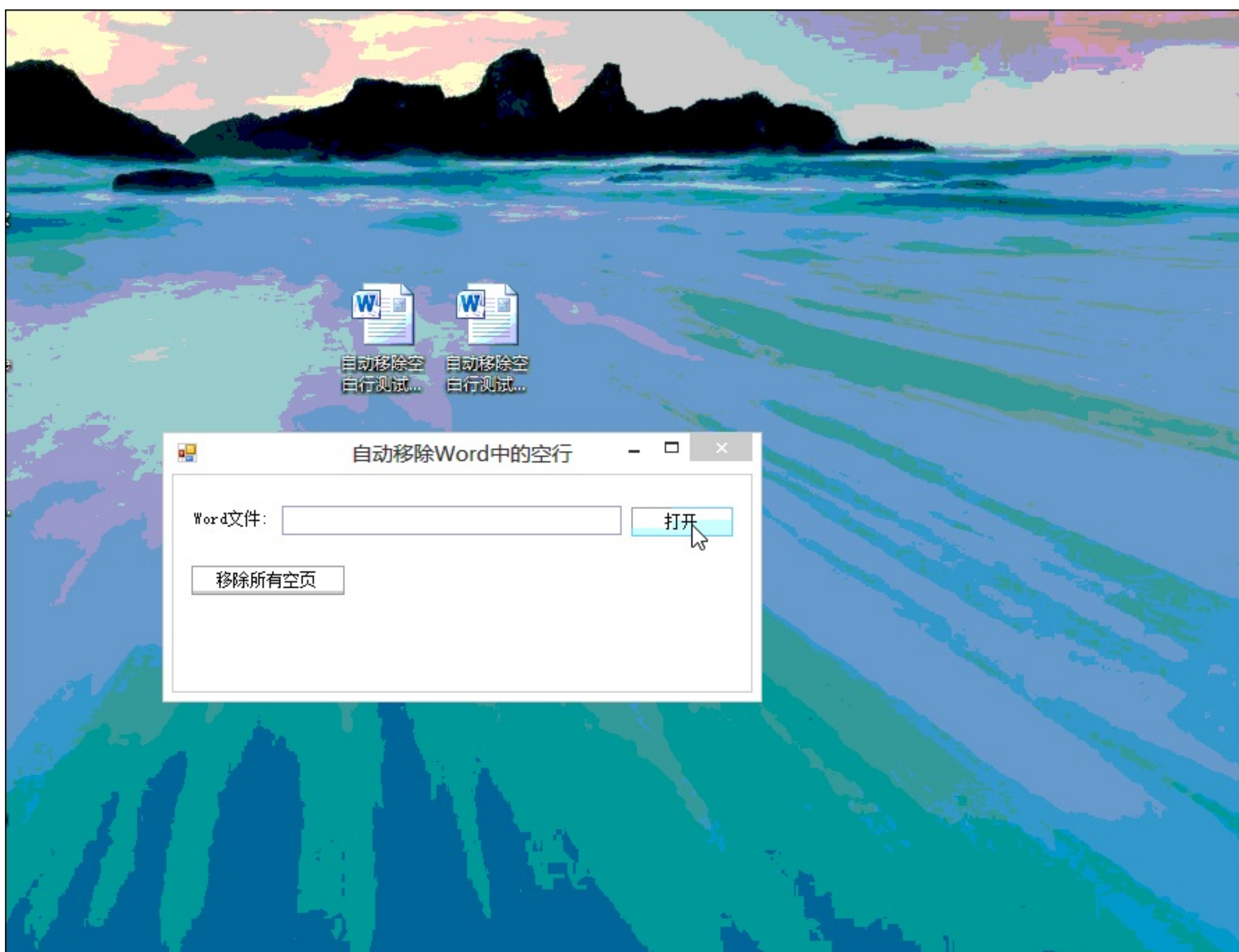
                MessageBox.Show("删除空白行成功");
            }
            catch (Exception ex)
```

```
            {
                MessageBox.Show("异常发生，异常信息为：" + ex.Message)
            }
            finally
            {
                // 释放资源
                // 退出Word程序
                if (wordapp != null)
                {
                    ((Word._Application)wordapp).Quit();
                }
                doc = null;
                wordapp = null;
            }
        }
```

为了测试该程序的正确性，这里我建立了两个测试文档，为了测试，我故意在文档中删除了空白行和空白页面，下面是两个测试文档的截图：



下面就看看该工具的运行效果(效果图是一段动画，认为这样可以更加说明运行效果)



:

三、人事部门的福音——自动给本月寿星员工发送邮件提醒

为了帮助大家更好地理解该程序，还是像之前一样，首先说说实现该程序的思路：

- 我们首先需要打开员工信息表，此时我们可以利用Excel对象模型中的**Excel.Application.Workbooks.Open**方法来获得一个工作簿对象，关于更多Excel对象模型的内容可以转向——创建Excel解决方案。
- 通过第一步我们已经获得了工作簿对象了，然后通过遍历工作簿中的激活表，即表格一(Sheet1),我们可以通过**workbook.ActiveSheet**来获得表格一对象。
- 遍历表格一中的所有行来找到生日信息中的月份，如果月份等于当前月份，就给该员工的邮箱进行发邮件。
- 对于自动发送邮件的实现，该实现和我们手动操作Outlook过程是一样，手动操作时，我们需要手动打开Outlook(在程序中就是创建**Outlook**应用程序对象)，然后点击新建邮件(在程序中就是通过**Applicatin**对象的**CreateItem(Outlook.OlItemType.olMailItem)**方法来创建一个邮件项目)，在新建邮件窗口中指定收件人，主题，邮件内容之后，点击Outlook中的发送邮件按钮(在程序中就是通过指定 **Outlook.MailItem**对象(即代表一个邮件窗体)的**To**(收件人)、**Subject**(主题)、**Body**(邮件内容)属性，然后再调用Send方法来发送邮件)

明白了思路之后，我们理解代码会更加容易了，具体实现代码为：

```
using System;
```

```
using System.IO;
using System.Runtime.InteropServices;
using System.Windows.Forms;

// 引用Excel和Outlook的命名空间
using Excel = Microsoft.Office.Interop.Excel;
using Outlook = Microsoft.Office.Interop.Outlook;

        string excelpath = string.Empty;
        // 打开员工表格
        private void btnOpen_Click(object sender, EventArgs e)
        {
            using (OpenFileDialog openFileDialog = new OpenFileDial
            {
                openFileDialog.Filter = "Excel File(*.xls;*.xlsx)|
                if (openFileDialog.ShowDialog() == DialogResult.OK
                {
                    txtExcelPath.Text = openFileDialog.FileName;
                    excelpath = openFileDialog.FileName;
                }
            }
        }

        // 自动给本月寿星发邮件通知
        private void btnSendEmail_Click(object sender, EventArgs e
        {
            if (!File.Exists(txtExcelPath.Text))
            {
                MessageBox.Show("员工表路径不存在，请确保输入正确的文件路
                return;
            }
            if (txbBirthday.Text.Trim() == string.Empty || txbEmail
            {
                MessageBox.Show("请先输入员工表中生日信息所在的列和邮箱信
                return;
            }

            // 输入信息都正确时开始发送邮件
            SendEmail(int.Parse(txbBirthday.Text.Trim()), int.Parse
        }
        // 发送邮件方法
        private void SendEmail(int birthDayColumn,int emailColumn)
        {
              // 获得当前月份
            int nowmonth = DateTime.Now.Month;
            // 发送邮件地址字符串
            string toEmailString = string.Empty;
            string emailBody="请收到邮件的员工，请本月28号到休闲室来参加生
            Excel.Application excelApp = null;
            Excel.Workbook workbook =null;
            Excel.Worksheet worksheet = null;
            Excel.Range range = null;
            try
```

```
            {
                // 新建Excel应用程序被设置它不可见
                excelApp = new Excel.Application();
                excelApp.Visible = false;
                workbook = excelApp.Workbooks.Open(excelpath);
                // 获得打开文件的激活表格
               worksheet= workbook.ActiveSheet;

                // 遍历表格中的所有行
                for (int row = 2; row < worksheet.UsedRange.Rows.Co
                {
                    // 因为我的测试表格中第四列是生日信息,在Excel中第一行的
                    // 这里本来需要在页面设置一个文本框让用户填写生日信息是
                    // 这里为了测试就直接在程序中指定
                    // 下面的Range就代表生日列中每一个单元格
                    range = worksheet.Cells[row, birthDayColumn];

                    // 我们可以通过Range.Value来获得单元格中的生日信息
                    // 因为我生日单元格中为日期格式, 所以获得的是日期类型,
                    int month = range.Value.Month;
                    // 如果我们的Excel文档中生日时间设置为文本格式的话, 这
                    // 通过Split函数来把生日信息以'/'符号分隔, 分隔的数组中
                    //int month = Int32.Parse(birthday.Split('/')[1
                    // 如果月份等于当前月的话, 就给这个人发邮件
                    if (month == nowmonth)
                    {
                        // 获得本月生日员工的邮件地址
                        toEmailString += ";" + ((Excel.Range)worksh
                    }
                }
            }
            catch (Exception ex)
            {
                MessageBox.Show("读取员工表格时出错, 异常信息为: " + ex.
                return;
            }
            finally
            {
                workbook.Close(Excel.XlSaveAction.xlDoNotSaveChange
                excelApp.Quit();
                if (workbook != null)
                {
                    Marshal.FinalReleaseComObject(workbook);
                    workbook = null;
                }
                if (excelApp != null)
                {
                    Marshal.FinalReleaseComObject(excelApp);
                    excelApp = null;
                }
            }

            if (CreateEmailItem("生日提醒", toEmailString, emailBody
```

```
            {
                MessageBox.Show("成功给本月寿星发送邮件提醒");
            }
        }
        // 创建邮件项
        private bool CreateEmailItem(string subjectEmail,string toE
        {
            Outlook.Application outlookapp = null;
            Outlook.MailItem email =null;

            try
            {
                // 创建邮件项，就如你手动点新建邮件一样
                outlookapp = new Outlook.Application();
                email = outlookapp.CreateItem(Outlook.OlItemType.ol
                // 指定邮件的主题，收件人和内容，就如你在新建邮件窗体中输入4
                email.Subject = subjectEmail;
                email.To = toEmail;
                email.Body = bodyEmail;
                email.Importance = Outlook.OlImportance.olImportanc

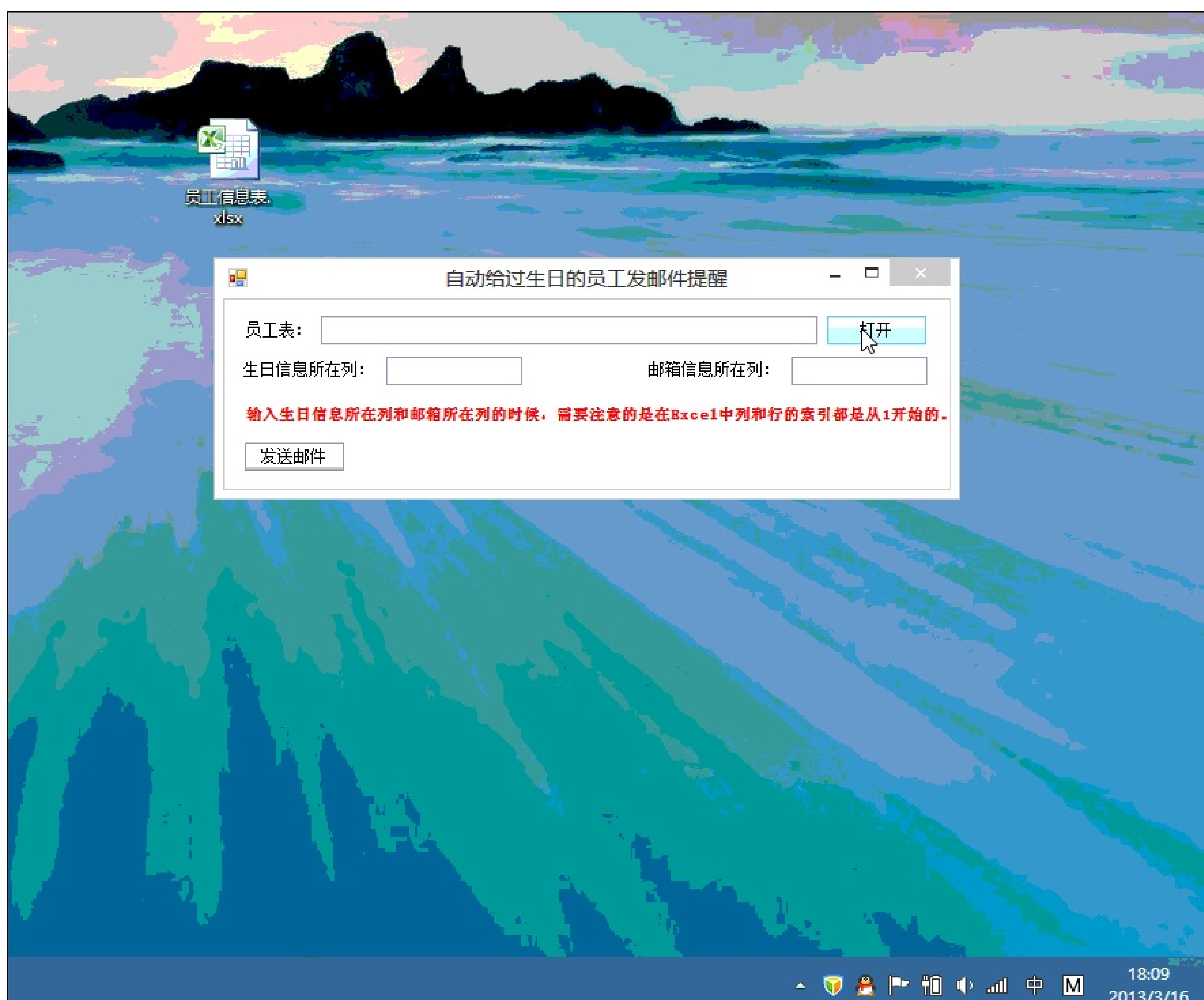
                // 发送邮件，就如你点界面上的发送邮件操作一样
                ((Outlook._MailItem)email).Send();
            }
            catch(Exception ex)
            {
                MessageBox.Show("发送邮件的时候失败，异常信息为：" + ex.
                return false;
            }
            finally
            {
                // 释放资源
                ((Outlook._Application)outlookapp).Quit();
                if (email != null)
                {
                    Marshal.FinalReleaseComObject(email);
                    email = null;
                }
                if (outlookapp != null)
                {
                    Marshal.FinalReleaseComObject(outlookapp);
                    outlookapp = null;
                }
            }

            return true;
        }
```

为了测试程序，我新建了一个员工信息表，表格的格式如下(你当然可以根据自己的需要更改格式)：

	A	B	C	D	E
1	姓名	性别	部门	生日	邮件地址
2	张三	男	研发部	1990/8/24	1234567@qq.com
3	李四	男	人事部	1988/6/12	794170314@qq.com
4	王五	女	销售部	1985/6/3	794170314@qq.com
5	赵六	女	研发部	1989/9/26	794170314@qq.com
6	刘八	男	人事部	1991/3/18	794170314@qq.com
7	LearningHard	男	研发部	1989/3/6	794170314@qq.com
8					
9					

现在就让我们看看该程序的运行效果：



四、 小结

到这里，本专题的内容就和大家介绍完了，在下一个专题中将向大家介绍下如何通过Office提供的API的来遥控幻灯片。如果大家对本专题中两个工具的实现源码有任何的疑问，都可以在下面留言给我。觉得不错的话，帮忙推荐下，感谢大家的支持

[C# 开发技巧系列] 使用C#操作幻灯片

本专题概要

- 引言
- 实现思路
- 遥控幻灯片程序的实现
- 小结

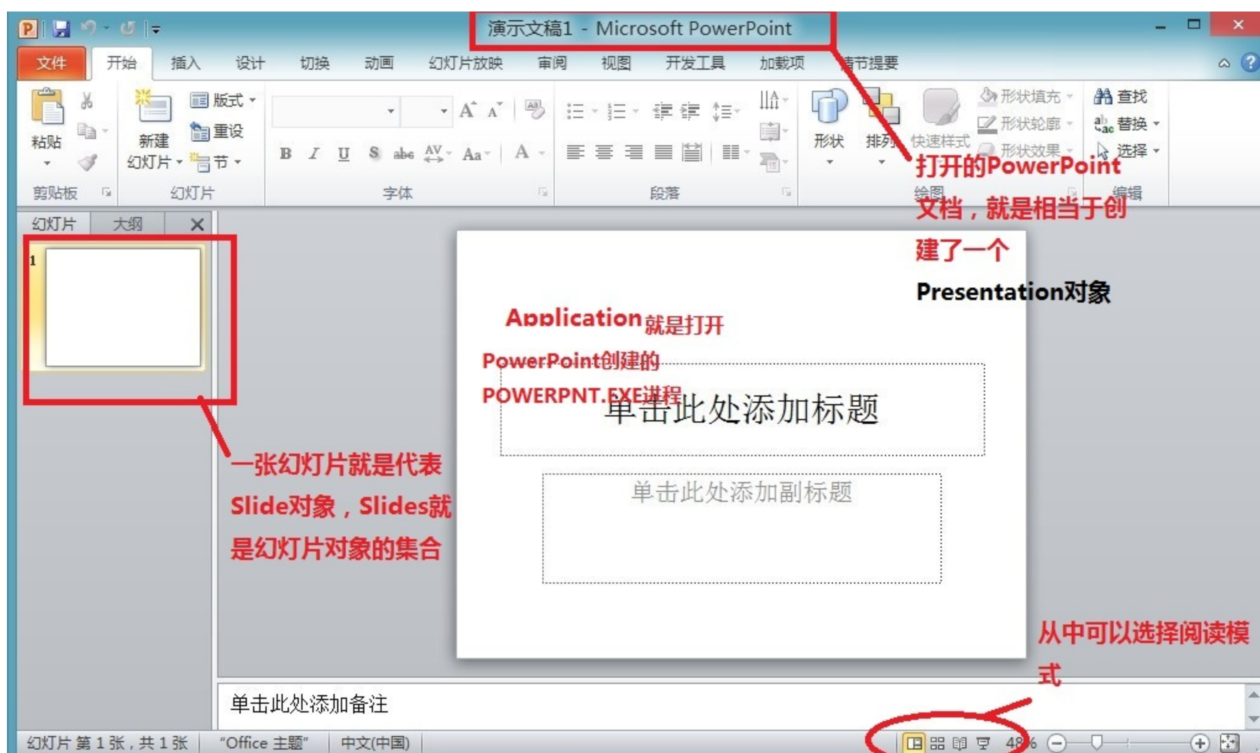
一、引言

记得老师讲课的时候，经常会用PPT遥控翻页笔来遥控幻灯片来给我们讲课，当时觉得非常有趣，由于这段时间接触了VSTO相关的开发，了解到了Office的相关产品都公开了一些API来让我们对Office产品进行二次开发，这时候我就想，能不能用PowerPoint公开的对象来制作一个遥控幻灯片的程序呢？在本专题就向大家介绍下这个小工具的实现思路和效果。

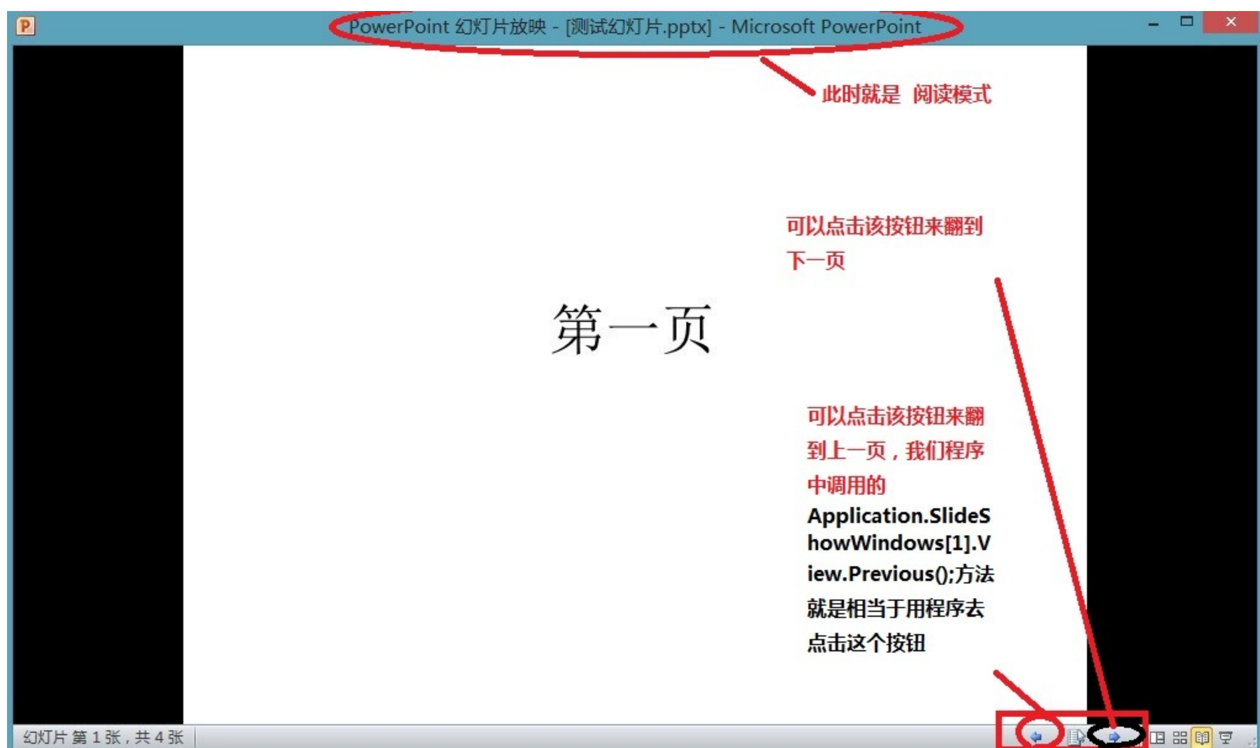
二、实现思路

1. 既然要实现的程序是遥控幻灯片，这样我们就需要先获得幻灯片应用程序的，在**PowerPoint**对象模型中，**Microsoft.Office.Interop.PowerPoint.Application**代表**Powerpoint**应用程序，这点和Word、Excel和Outlook都是一样的。
2. 获得了幻灯片应用程序对象之后，之后我们就需要获得幻灯片对象，因为我们遥控的是幻灯片，在PowerPoint对象模型中也提供了幻灯片对象，即**Microsoft.Office.Interop.PowerPoint.Slide**。由于幻灯片又是存在于演示文稿中的，所以我们要想获得幻灯片对象，就需要先获得演示文稿对象，**Microsoft.Office.Interop.PowerPoint.Presentation** 就是代表演示文稿对象。
3. 获得幻灯片对象之后，我们就可以利用幻灯片对象的**Select**方法来进行幻灯片的切换,然而在阅读模式的情况下，不能用**Select**方法来进行翻页，此时需要另一种方式来实现，即调用**Microsoft.Office.Interop.PowerPoint.SlideShowView**对象的**First, Next,Last,Previous**方法来进行幻灯片翻页。

上面列出来的就是该工具的实现思路，其实思路非常的简单，为了帮助大家更形象地理解PowerPoint的对象模型，下面就用一张图来介绍PowerPoint中对象与真真的幻灯片的一个对象关系（从下面的图中也可以体会到面向对象编程，就是把看到的东西抽象出一个个对象）：



下面一张是阅读模式下程序中实现翻页功能与在幻灯片中的对应关系：



三、遥控幻灯片程序的实现

有了上面的解释，我们再看下面的实现代码时，相信大家肯定不会觉得有任何难道了，下面就直接贴出部分的实现的代码(这样可以让大家参看代码自己去实现剩余的部分，当然在文章的最后也会提供全部源码的下载)：

```
/// <summary>
/// 检查是否打开幻灯片程序
```

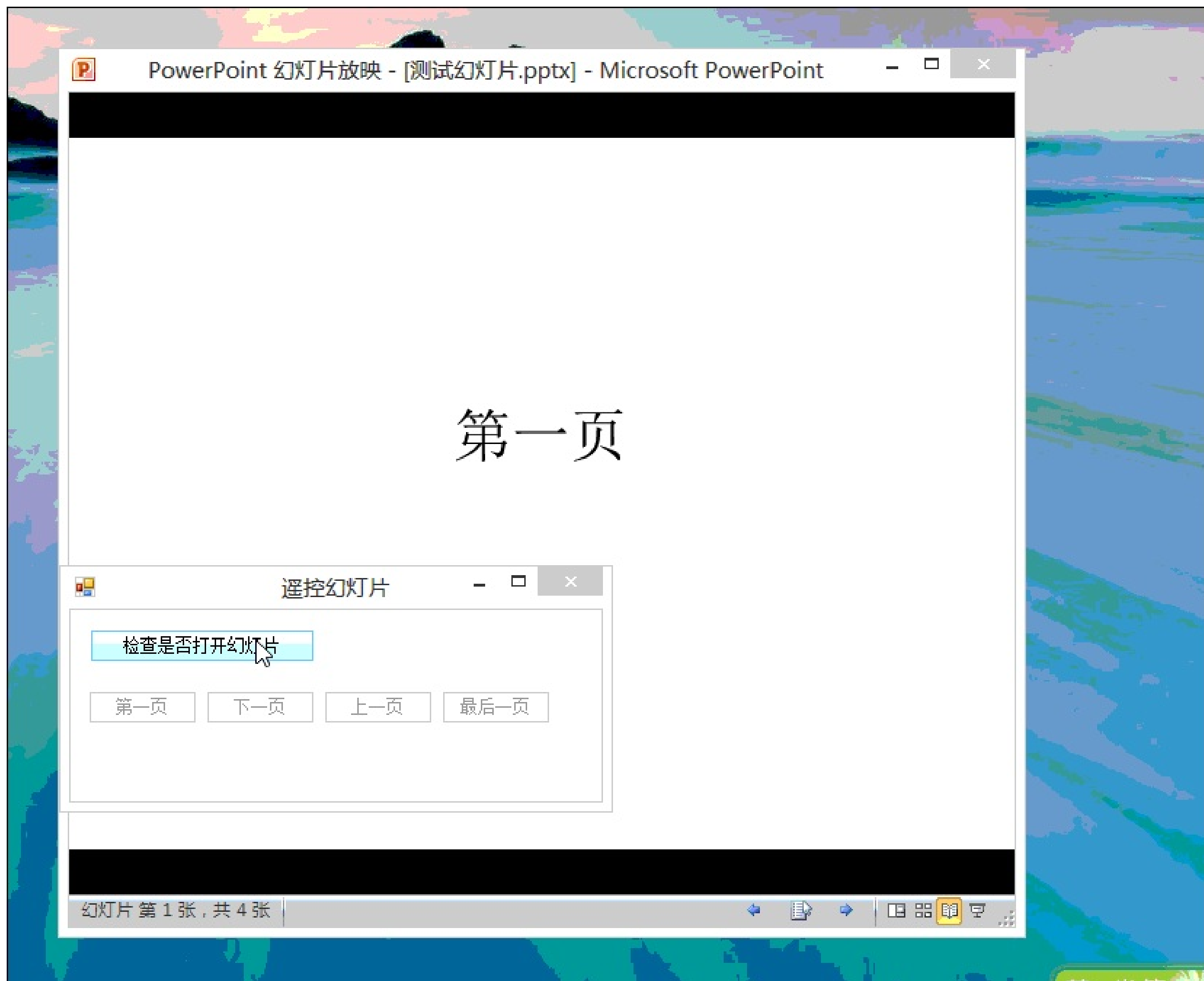
```
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void btnCheck_Click(object sender, EventArgs e)
{
    // 必须先运行幻灯片，下面才能获得PowerPoint应用程序，否则会出现异常
    // 获得正在运行的PowerPoint应用程序
    try
    {
        pptApplication = Marshal.GetActiveObject("PowerPoint.Applic

        // 成功获取了PowerPoint程序时，使UI按钮可用
        this.btnFirst.Enabled = true;
        this.btnNext.Enabled = true;
        this.btnPrev.Enabled = true;
        this.btnLast.Enabled = true;
    }
    catch
    {
        MessageBox.Show("请先启动遥控的幻灯片", "Error", MessageBoxBut
    }
    if (pptApplication != null)
    {
        //获得演示文稿对象
        presentation = pptApplication.ActivePresentation;
        // 获得幻灯片对象集合
        slides = presentation.Slides;
        // 获得幻灯片的数量
        slidescount = slides.Count;
        // 获得当前选中的幻灯片
        try
        {
            // 在普通视图下这种方式可以获得当前选中的幻灯片对象
            // 然而在阅读模式下，这种方式会出现异常
            slide = slides[pptApplication.ActiveWindow.Selection.Sl
        }
        catch
        {
            // 在阅读模式下出现异常时，通过下面的方式来获得当前选中的幻灯片对
            slide = pptApplication.SlideShowWindows[1].View.Slide;
        }
    }
}

// 第一页事件
private void btnFirst_Click(object sender, EventArgs e)
{
    try
    {
        // 在普通视图中调用Select方法来选中第一张幻灯片
        slides[1].Select();
        slide = slides[1];
    }
```

```
        catch
        {
            // 在阅读模式下使用下面的方式来切换到第一张幻灯片
            pptApplication.SlideShowWindows[1].View.First();
            slide = pptApplication.SlideShowWindows[1].View.Slide;
        }
    }
```

下面就看看该遥控程序的运行效果是如何的:



四、小结

到这里本专题的介绍就介绍就结束，其实本程序最好是放在手机客户端，这样我们就可以利用手机来对我们的幻灯片进行翻页了，这样就和激光笔的效果就是一样的了，这里就给大家先提供一个思路吧，我相信如果要在手机客户端实现的话，肯定就需要蓝牙编程的技术或者WiFi编程的技术来获取笔记本电脑的幻灯片应用程序，只要我们成功在手机客户端获取了PowerPoint应用程序对象的话，后面的实现过程就和本程序的实现方式就基本一样的，然而我们同时打开笔记本的蓝牙和手机的蓝牙(也可以利用WiFi)，这样我们就可以轻松实现用手机来遥控我们演讲文稿了，如果有时间的话，也会研究下手机的蓝牙编程技术，实现了肯定会在博客中向大家分享的，如果其他朋友提前实现了的，也不要忘记在博客分享给大家了。

程序所有源码：http://code.msdn.microsoft.com/PowerPoint-42854d28#content
（麻烦大家下载的时候帮忙点下评级）

[C# 开发技巧系列]如何动态设置屏幕分辨率

因为最近在MSDN论坛和stackflow中看到一些朋友经常问到这个问题，所以写这篇文章来帮助大家遇到相同问题的时候可以很快的得到解决，下面就不啰嗦了，直接看代码如何解决这个问题的。

首先，大家应该明确，现在没有可用的API来给我们动态地设置屏幕分辨率，我们要实现这个需求，我们只能在**C#**程序中调用**Win32 API** 函数来解决这个问题的，这里用C#代码调用Win32 API 就涉及到一个问题的，即.NET 互操作性的问题，关于这个大家可以参考我的互操作性系列文章。这里我就不过多解释了。

我们要解决这个问题，首先大家肯定也会遇到一个经常遇到的问题，即如何获得用户的分辨率，对于这个问题，.NET中提供的单独的类给我们调用，我们可以使用Screen这个类，具体看下面的示例代码：

然后就是如何改变屏幕的分辨率呢？要更改显示设置可以通过使用两个 Win32 API 来完成，这两个 API 都具有指向 DEVMODE.aspx) 结构的指针，它们分别包含与显示设置有关的所有信息：

- 使用 EnumDisplaySettings.aspx) 读取当前显示设置，并枚举所有受支持的显示设置。
- 使用 ChangeDisplaySettings.aspx) 切换到新的显示设置。

第一步、我们要先定义DEVMODE 结构体，该结构的结构必须与DEVMODE的结构一致，下面是C#中对DEVMODE 结构体的定义代码：

```
// 映射 DEVMODE 结构
    // 可以参照 DEVMODE结构的指针定义：
    // http://msdn.microsoft.com/en-us/library/windows/desktop/dd18
    [StructLayout(LayoutKind.Sequential, CharSet = CharSet.Ansi)]
    public struct DEVMODE
    {
        [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 32)]
        public string dmDeviceName;

        public short dmSpecVersion;
        public short dmDriverVersion;
        public short dmSize;
        public short dmDriverExtra;
        public int dmFields;
        public int dmPositionX;
        public int dmPositionY;
        public int dmDisplayOrientation;
        public int dmDisplayFixedOutput;
        public short dmColor;
        public short dmDuplex;
        public short dmYResolution;
        public short dmTTOption;
        public short dmCollate;

        [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 32)]
        public string dmFormName;

        public short dmLogPixels;
        public short dmBitsPerPel;
        public int dmPelsWidth;
        public int dmPelsHeight;
        public int dmDisplayFlags;
        public int dmDisplayFrequency;
        public int dmICMMethod;
        public int dmICMIntent;
        public int dmMediaType;
        public int dmDitherType;
        public int dmReserved1;
        public int dmReserved2;
        public int dmPanningWidth;
        public int dmPanningHeight;
    };
```

View Code

第二步、在托管环境下对Win 32 函数进行声明：

```
// Win32 函数在托管环境下的声明
    public class  NativeMethods
    {
        // 平台调用的申明
        [DllImport("user32.dll")]
        public static extern int EnumDisplaySettings(
          string deviceName, int modeNum, ref DEVMODE devMode);
        [DllImport("user32.dll")]
        public static extern int ChangeDisplaySettings(
              ref DEVMODE devMode, int flags);

        // 控制改变屏幕分辨率的常量
        public const int ENUM_CURRENT_SETTINGS = -1;
        public const int CDS_UPDATEREGISTRY = 0x01;
        public const int CDS_TEST = 0x02;
        public const int DISP_CHANGE_SUCCESSFUL = 0;
        public const int DISP_CHANGE_RESTART = 1;
        public const int DISP_CHANGE_FAILED = -1;

        // 控制改变方向的常量定义
        public const int DMDO_DEFAULT = 0;
        public const int DMDO_90 = 1;
        public const int DMDO_180 = 2;
        public const int DMDO_270 = 3;
    }
```

View Code

第三步、调用EnumDisplaySettings.aspx)和ChangeDisplaySettings.aspx)这两个函数来实现动态改变屏幕分辨率，具体代码如下：

```
    // 改变分辨率
        public ChangeResolution(int width, int height)
        {
            // 初始化 DEVMODE结构
            DEVMODE devmode = new DEVMODE();
            devmode.dmDeviceName = new String(new char[32]);
            devmode.dmFormName = new String(new char[32]);
            devmode.dmSize = (short)Marshal.SizeOf(devmode);

            if (0 != NativeMethods.EnumDisplaySettings(null, Native
            {
                devmode.dmPelsWidth = width;
                devmode.dmPelsHeight = height;

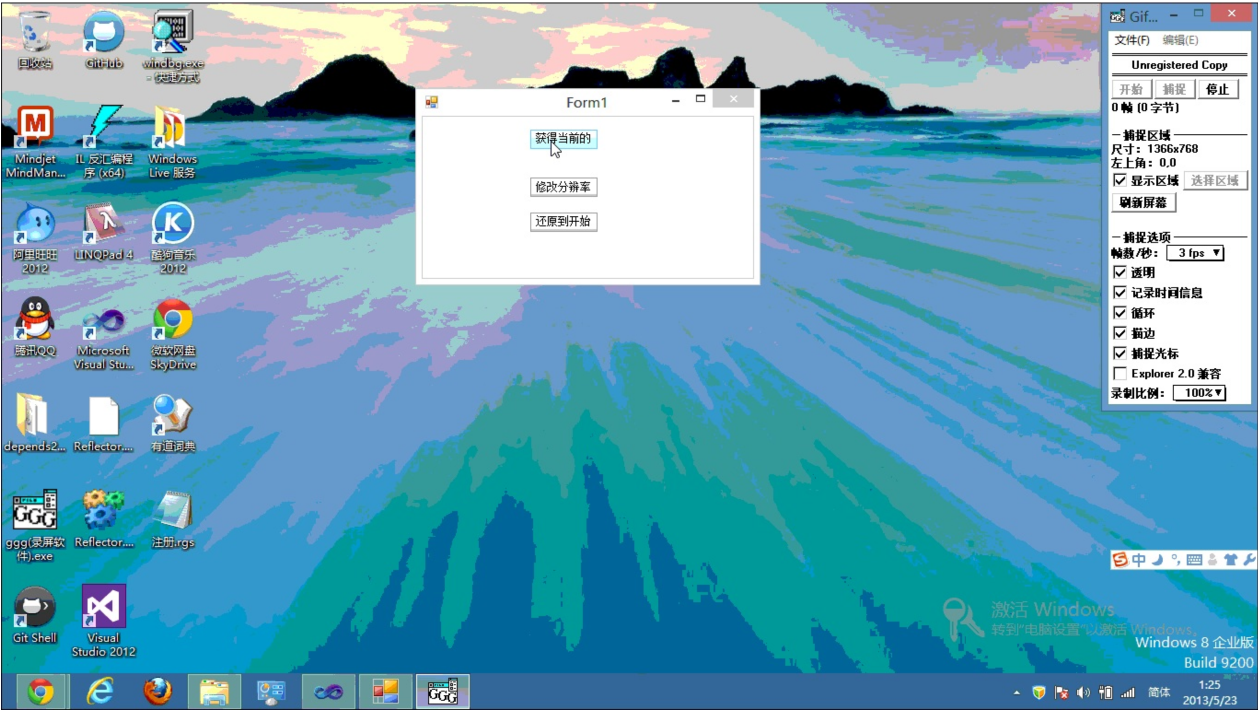
                // 改变屏幕分辨率
                int iRet = NativeMethods.ChangeDisplaySettings(ref

                if (iRet == NativeMethods.DISP_CHANGE_FAILED)
                {
                    MessageBox.Show("不能执行你的请求", "信息", Messag
                }
                else
                {
                    iRet = NativeMethods.ChangeDisplaySettings(ref

                    switch (iRet)
                    {
                            // 成功改变
                        case NativeMethods.DISP_CHANGE_SUCCESSFUL:
                            {
                                break;
                            }
                        case NativeMethods.DISP_CHANGE_RESTART:
                            {
                                MessageBox.Show("你需要重新启动电脑设置
                                break;
                            }
                        default:
                            {
                                MessageBox.Show("改变屏幕分辨率失败",
                                break;
                            }
                    }
                }
            }
        }
```

View Code

为了大家更加形象地看到程序的运行结果，下面是一个演示效果：



实现源代码下载：[动态设置屏幕分辨率.zip]

[C# 开发技巧系列]C#如何实现图片查看器

本专题概要

- 一、引言
- 二、实现思路
- 三、实现效果
- 四、小结

一、引言

因为最近在MSDN中的论坛和CSDN论坛都看到有些朋友问到如何用C#实现一个像Windows自带的图片查看器的功能等类似的问题(当然还有如何如何旋转图片的，如何通过按钮来变换图片的功能等)，所以为了帮助大家更好地解决类似的这样的问题，所以这篇文章将简单介绍下如何使用C#来实现一个图片查看器的功能的，该工具保存的功能有：

1. 可以通过“上一张” “下一张”这样的按钮来轮换浏览图片
2. 实现对图片的旋转
3. 实现对旋转后图片的保存功能。本程序不仅提供旋转**90/180/270**这样的实现，同时提供一个方法来完成旋转任意角度的实现
4. 该程序未实现**Windows**图片查看图片缩放的功能，这部分的功能主要要点是改变图片在**PictureBox**控件中的高度和宽度就可以的

二、实现思路

2.1 图片轮换浏览功能的实现

首先分析下第一个功能点的实现,要实现图片的轮换浏览,我们可以根据下面的思路来实现：

- 第一步、获得目录下所有图片的集合，此时使用**Directory.GetFiles()**来获得目录下所有文件，然后再对该集合进行筛选，筛选出是图片的文件，代码用扩展名进行筛选的
- 第二步、获得所有图片集合之后，实现图片轮换就需要改变这个集合的索引就可以实现上一张和下一张的功能了
- 第三步、需要考虑到最后一张或者第一张的情况下，再点击下一张或上一张图片来轮换成第一张或最后一张

思路就是上面的，有了上面的思路之后，就让我们看看具体的代码来对照理解下:

```
**// 第一步** // 获得预览图片文件路径下的图片集合
```

```
        public static List<string> GetImgCollection(string path)
        {
            string[] imgarray = Directory.GetFiles(path);
            var result = from imgstring in imgarray
                         where imgstring.EndsWith("jpg", StringComp
                         imgstring.EndsWith("png", StringComparison
                         imgstring.EndsWith("bmp", StringComparison
                         select imgstring;
            return result.ToList();
        }

         **// 第二步** // 获得打开图片在图片集合中的索引
        private int GetIndex(string imagepath)
        {
            int index = 0;
            for (int i = 0; i < imgArray.Count; i++)
            {
                if (imgArray[i].Equals(imagepath))
                {
                    index = i;
                    break;
                }
            }

            return index;
        }

        // 切换图片的方法
        private void SwitchImg(int index)
        {
            newbitmap = Image.FromFile(imgArray[index]);
            picBoxView.Image = newbitmap;
            imgPath = imgArray[index];
        }

        **// 第三步** // 上一张图片
        private void btnPre_Click(object sender, EventArgs e)
        {
            int index = GetIndex(imgPath);
            // 释放上一张图片的资源，避免保存的时候出现ExternalException异
            newbitmap.Dispose();
            if (index == 0)
            {
                SwitchImg(imgArray.Count - 1);
            }
            else
            {
                SwitchImg(index - 1);
            }
        }

        // 下一张图片
        private void btnNext_Click(object sender, EventArgs e)
```

```
        {
            int index = GetIndex(imgPath);
            // 释放上一张图片的资源，避免保存的时候出现ExternalException异
            // 经常在调用Save方法的时候都会出现 一个GDI一般性错误,主要原因是
            newbitmap.Dispose();
            if (index != imgArray.Count - 1)
            {
                SwitchImg(index + 1);
            }
            else
            {
                SwitchImg(0);
            }
        }
```

2.2 图片旋转功能的实现

上面的代码实现了第一个功能点的问题了,下面就解释下如何实现第二个功能点——图片旋转的问题:

对于Windows自带的图片查看器，它旋转的角度只能顺时针旋转90或逆时针旋转90度，这个功能实现起来可以说非常简单，只需要使用**Image.RotateFlip(RotateFlipType)**方法就可以完成的，有些朋友也想对图片实现旋转任意角度，对于这个问题源码中也有具体的实现，大家可以从文章的最后下载源码进行查看，这里就不贴出具体代码的，下面就看看如何实现Windows自带的图片查看器的旋转功能的代码：

```
    // 顺时针旋转90度旋转图片
        private void btnRotate_Click(object sender, EventArgs e)
        {
            picBoxView.SizeMode = PictureBoxSizeMode.Zoom;

            // 顺时针旋转90度的另外一种实现
            newbitmap.RotateFlip(RotateFlipType.Rotate90FlipNone);
            picBoxView.Image = newbitmap;
            isRotate = true;
            //newbitmap = (Image)ImageManager.RotateImg(bitmap, 90f
            //bitmap.Dispose();
            //picBoxView.Image = newbitmap;
        }

        // 逆时针旋转90度
        private void btncounterclockwiseRotate_Click(object sender,
        {
            picBoxView.SizeMode = PictureBoxSizeMode.Zoom;

            // 逆时针旋转90度的另外实现
            newbitmap.RotateFlip(RotateFlipType.Rotate270FlipNone);
            picBoxView.Image = newbitmap;
            isRotate = true;
            // 下面是旋转任意角度的代码
            //newbitmap = (Image)ImageManager.RotateImg(bitmap, 360
            //bitmap.Dispose();
            //picBoxView.Image = newbitmap;
        }
```

2.3 对旋转图片的保存功能的实现

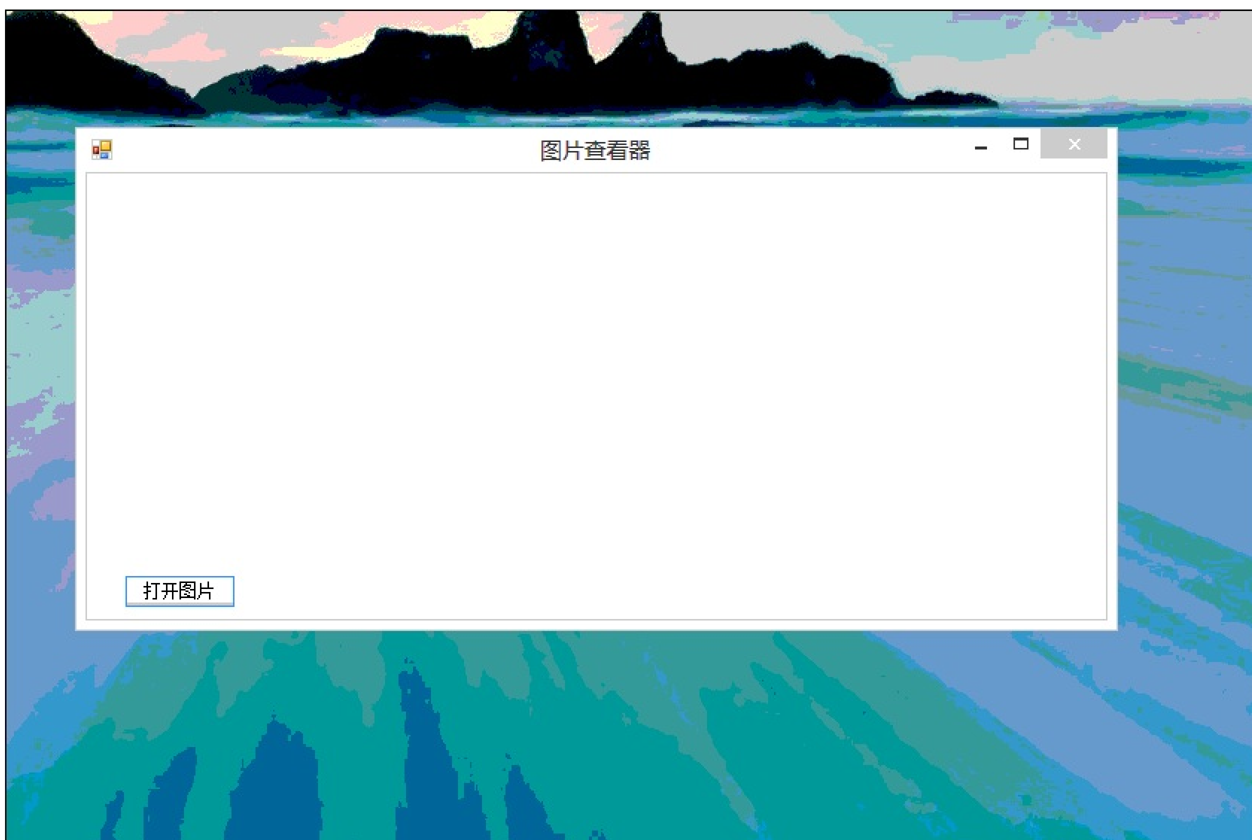
最后就是针对旋转图片保存的实现了,此时我参考了Windows自带图片查看器的实现方式,因为我用Windows自带图片查看器浏览图片的实现，当我旋转图片时，它并不是实时地保存到旋转的图片的，而是当我关闭Windows自带图片查看器的时候，旋转的图片才保存到文件中的，有了这个思路之后，我就把我保存的代码逻辑放在窗体的关闭的事件处理程序中来实现的，此时保存的功能我们只需要调用**Image.Save(path)**方法就可以完成对图片的保存，下面就看看具体代码的实现的：

```
// 关闭窗体后保存旋转后的图片到文件中
        private void Form1_FormClosed(object sender, FormClosedEven
        {
            if (imgPath == null || isRotate == false)
            {
                return;
            }

            // 保存旋转后的图片
            switch (Path.GetExtension(imgPath).ToLower())
            {
                case ".png":
                    newbitmap.Save(imgPath, ImageFormat.Png);
                    newbitmap.Dispose();
                    break;
                case ".jpg":
                    newbitmap.Save(imgPath);
                    newbitmap.Dispose();
                    break;
                default:
                    newbitmap.Save(imgPath, ImageFormat.Bmp);
                    newbitmap.Dispose();
                    break;
            }
        }
```

三、实现效果

上面已经介绍了实现该程序的一个思路的，朋友是不是迫不及待的想看到到底自定义图片查看器是什么样子的呢？下面就通过一个动画来让大家更形象地看到程序的运行效果的：



四、小结

到这里该文章的内容就介绍结束了,希望大家如果遇到类似的问题可以很快从这篇博客中得到解决,另外附带下MSDN中这个问题的链接:

http://social.msdn.microsoft.com/Forums/zh-CN/visualcshartzhchs/thread/89d09d59-ab82-4e41-896f-daab68edbd10

本专题源码下载:图片查看器

[C# 开发技巧]如何防止程序多次运行

一、引言

最近发现很多人在论坛中问到如何防止程序被多次运行的问题的,如: http://social.msdn.microsoft.com/Forums/zh-CN/6398fb10-ecc2-4c03-ab25-d03544f5fcc9, 所以这里就记录下来，希望给遇到同样问题的朋友有所参考的,同时也是对自己的一个积累。在介绍具体实现代码之前，我们必须明确解决这个问题的思路是什么的？下面只要分享我的一个思考的这个问题的方式：

1. 当我们点击一个exe文件时，此时该exe程序将会运行，我们可以看到该程序的界面，对于计算机而言，就是会在系统上开启一个该程序的进行，这个我们可以通过任务管理器来查看的（当我们点击exe之后，程序运行，系统会创建一个与与程序同名的进程）
2. 既然我们要防止程序运行多次，也就是说程序只能运行一次，从操作系统的角度来讲就是该程序的进程只能是唯一的，分析到这里我们自然就想到了，要保证该程序进程只有一个，我们就要判断下该程序进程是否在自己的操作系统上运行了,如果已经运行了一个进程，当我们下次运行**exe**的时候，此时不是再开启该程序进程，而是退出，弹出一个提示框告诉用户该程序已经运行，如果操作系统没有运行该程序进程，则运行这个程序。
3. 从而这个问题就转换为判断该程序进程的数量问题了，此时我们就想.NET 有没有提供一个类可以获得该进程名的数量，如果数量大于1则说明该程序已经运行了，小于就表明程序没有运行。如果熟悉.NET类库的人肯定知道.NET类库中有一个Process类，该类的意思就是一个进程的抽象。(有些人就会说，我一开始不知道有这个类那怎么办呢？那就是考验你英文了，因为进程的英文就是Process，然而所有编程语言的命名都很通俗易懂，此时就可以用Process在MSDN上搜索，这样你也就发现这个类了)
4. 除了第三点中提出找进程数量的思路外，还有另外一种实现思路就是——我们能不能让运行一个进程的时候，让该进程具有一个变量，该变量是唯一标识该进程，当点击**exe**文件预创建一个改程序进程时，我们去判断这个变量是否存在，如果存在就说明这个进程已经运行，从而退出本次的程序，并且提示给用户说该程序已经运行。

从上面的分析过程中可以看出，我们解决这个问题的思路就是从进程入手，第三点的思路就是直接从进程数量入手，而第四点思路也是从进程入手，只是做了一个变换罢了，让一个变量来唯一标识一个进程，当变量存在时说明该程序进程也运行了。

二、使用互斥量Mutex

弄懂了主要的实现思路之后，下面看代码实现就完全不是问题了，使用互斥量的实现就是第四点的思路的体现，我们用为该程序进程创建一个互斥量Mutex对象变量，当运行该程序时，该程序进程就具有了这个互斥的Mutex变量，如果再次运行该程序时，通过检查该互斥变量是否存在（来替换检测这个进程是否存在），如果

存在则说明程序已运行，否则就没运行。这里需要注意的是：从我的多线程同步的文章大家可以知道，Mutex类也可以对线程进行同步，那是不是其他对线程同步的类也可以解决本专题中的问题呢？答案是否定，之所以Mutex类可以解决这个问题，是因为Mutex类除了可以对线程同步，也可以对进程同步。下面就具体看看实现代码吧：

```
using System;
using System.Threading;
using System.Windows.Forms;

namespace OnlyInstanceRunning
{
    static class Program
    {
        /// <summary>
        /// 应用程序的主入口点。
        /// </summary>
        [STAThread]
        static void Main()
        {
            #region 方法一:使用互斥量
            bool createNew;

            //  createdNew:
            // 在此方法返回时，如果创建了局部互斥体（即，如果 name 为 null
            // 如果指定的命名系统互斥体已存在，则为false
            using (Mutex mutex = new Mutex(true, Application.Produc
            {
                if (createNew)
                {
                    Application.EnableVisualStyles();
                    Application.SetCompatibleTextRenderingDefault(f
                    Application.Run(new Form1());
                }
                // 程序已经运行的情况，则弹出消息提示并终止此次运行
                else
                {
                    MessageBox.Show("应用程序已经在运行中...");
                    System.Threading.Thread.Sleep(1000);

                    //  终止此进程并为基础操作系统提供指定的退出代码。
                    System.Environment.Exit(1);
                }
            }

            #endregion
        }
    }
}
```

三、直接判断进程是否存在的方式来解决这个问题

3.1 判断该程序进程数量的方式

有了上面的思路分析之后，相信大家看下面代码会觉得一目了然，这里就不多解释了，直接看代码：

```
    #region 方法二:使用进程名
            Process[] processcollection = Process.GetProcessesByNar
            // 如果该程序进程数量大于，则说明该程序已经运行，则弹出提示信息并
            if (processcollection.Length >= 1)
            {
                MessageBox.Show("应用程序已经在运行中。。");
                Thread.Sleep(1000);
                System.Environment.Exit(1);
            }
            else
            {
                Application.EnableVisualStyles();
                Application.SetCompatibleTextRenderingDefault(false
                // 运行该应用程序
                Application.Run(new Form1());
            }
            #endregion
```

3.2 直接判断程序进程是否存在的方式

```
using System;
using System.Diagnostics;
using System.Reflection;
using System.Runtime.InteropServices;
using System.Windows.Forms;

namespace Way3
{
    static class Program
    {
        #region 方法三：使用的Win32函数的声明
        /// <summary>
        ///  设置窗口的显示状态
        ///  Win32 函数定义为：http://msdn.microsoft.com/en-us/librar
        /// </summary>
        /// <param name="hWnd">窗口句柄</param>
        /// <param name="cmdShow">指示窗口如何被显示</param>
        /// <returns>如果窗体之前是可见，返回值为非零；如果窗体之前被隐藏，
        [DllImport("User32.dll")]
```

```
        private static extern bool ShowWindow(IntPtr hWnd, int cmdS

        /// <summary>
        ///  创建指定窗口的线程设置到前台，并且激活该窗口。键盘输入转向该窗口
        ///  系统给创建前台窗口的线程分配的权限稍高于其他线程。
        /// </summary>
        /// <param name="hWnd">将被激活并被调入前台的窗口句柄</param>
        /// <returns>如果窗口设入了前台，返回值为非零；如果窗口未被设入前台，
        [DllImport("User32.dll")]
        private static extern bool SetForegroundWindow(IntPtr hWnd)

        // 指示窗口为普通显示
        private const int WS_SHOWNORMAL = 1;
        #endregion

        /// <summary>
        /// 应用程序的主入口点。
        /// </summary>
        [STAThread]
        static void Main()
        {
            #region 方法三：调用Win32 API,并激活运行程序的窗口显示在最前端
            // 这种方式在VS调用的情况不成立的，因为在VS中按F5运行的进程为On
            // 关于这个进程的更多内容可以查看：http://msdn.microsoft.com
            // 而直接点OnlyInstanceRunning.exe运行的程序进程为OnlyInsta
            // 但是我们可以一些小的修改，即currentProcess.ProcessName.R

            // 获得正在运行的程序，如果没有相同的程序，则运行该程序
            Process process = RunningInstance();
            if (process == null)
            {
                Application.EnableVisualStyles();
                Application.SetCompatibleTextRenderingDefault(false
                Application.Run(new Form1());
            }
            else
            {
                // 已经运行该程序，显示信息并使程序显示在前端
                MessageBox.Show("应用程序已经在运行中......");
                HandleRunningInstance(process);
            }
            #endregion
        }

        #region 方法三定义的方法
        /// <summary>
        /// 获取正在运行的程序，没有运行的程序则返回null
        /// </summary>
        /// <returns></returns>
        private static Process RunningInstance()
        {
            // 获取当前活动的进程
            Process currentProcess = Process.GetCurrentProcess();
```

```
            // 根据当前进程的进程名获得进程集合
            // 如果该程序运行，进程的数量大于1
            Process[] processcollection = Process.GetProcessesByNar
            foreach (Process process in processcollection)
            {
                // 如果进程ID不等于当前运行进程的ID以及运行进程的文件路径等
                // 则说明同一个该程序已经运行了，此时将返回已经运行的进程
                if (process.Id != currentProcess.Id)
                {
                    if (Assembly.GetExecutingAssembly().Location.Re
                    {
                        return process;
                    }
                }
            }

            return null;
        }

        /// <summary>
        /// 显示已运行的程序
        /// </summary>
        /// <param name="instance"></param>
        private static void HandleRunningInstance(Process instance)
        {
            // 显示窗口
            ShowWindow(instance.MainWindowHandle, WS_SHOWNORMAL);

            // 把窗体放在前端
            SetForegroundWindow(instance.MainWindowHandle);
        }

        #endregion
    }
}
```

3.3 解决3.2实现方式中存在的问题——只能是最小化的窗体显示出来，如果隐藏到托盘中则不能把运行的程序显示出来

```
using System;
using System.Diagnostics;
using System.Runtime.InteropServices;
using System.Windows.Forms;

namespace Way4
{
```

```
    static class Program
    {

        #region 方法四：使用的Win32函数的声明

        /// <summary>
        /// 找到某个窗口与给出的类别名和窗口名相同窗口
        /// 非托管定义为：http://msdn.microsoft.com/en-us/library/win
        /// </summary>
        /// <param name="lpClassName">类别名</param>
        /// <param name="lpWindowName">窗口名</param>
        /// <returns>成功找到返回窗口句柄,否则返回null</returns>
        [DllImport("user32.dll")]
        public static extern IntPtr FindWindow(string lpClassName,

        /// <summary>
        /// 切换到窗口并把窗口设入前台,类似 SetForegroundWindow方法的功能
        /// </summary>
        /// <param name="hWnd">窗口句柄</param>
        /// <param name="fAltTab">True代表窗口正在通过Alt/Ctrl +Tab被
        [DllImport("user32.dll ", SetLastError = true)]
        static extern void SwitchToThisWindow(IntPtr hWnd, bool fAl

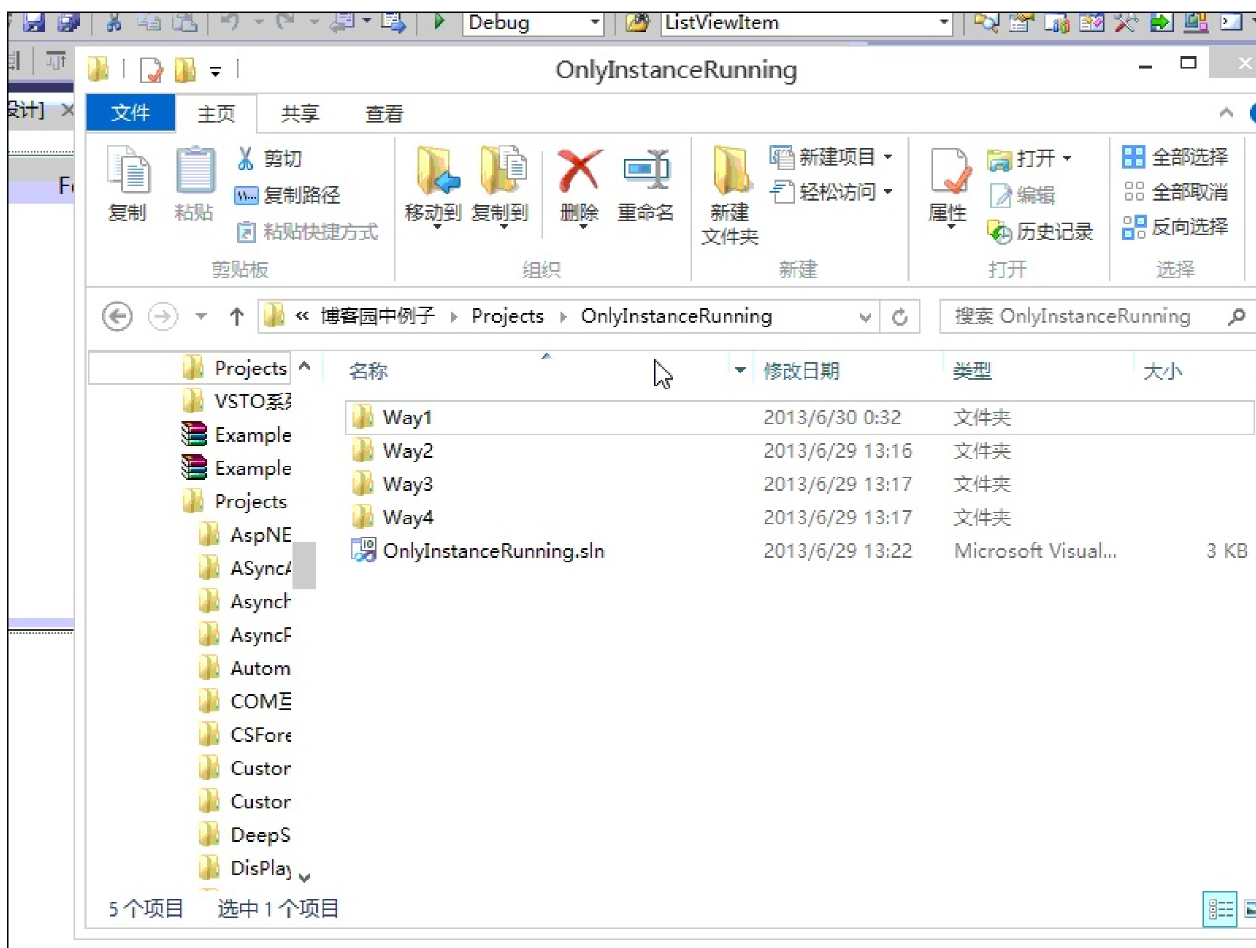
        ///// <summary>
        /////  设置窗口的显示状态
        /////  Win32 函数定义为：http://msdn.microsoft.com/en-us/libr
        ///// </summary>
        ///// <param name="hWnd">窗口句柄</param>
        ///// <param name="cmdShow">指示窗口如何被显示</param>
        ///// <returns>如果窗体之前是可见，返回值为非零；如果窗体之前被隐藏
        [DllImport("user32.dll", EntryPoint = "ShowWindow", CharSet
        public static extern int ShowWindow(IntPtr hwnd, int nCmdSh
        public const int SW_RESTORE = 9;
        public static IntPtr formhwnd;
        #endregion

        /// <summary>
        /// 应用程序的主入口点。
        /// </summary>
        [STAThread]
        static void Main()
        {
            #region 方法四: 可以是托盘中的隐藏程序显示出来
            // 方法四相对于方法三而言应该可以说是一个改进,
            // 因为方法三只能是最小化的窗体显示出来，如果隐藏到托盘中则不能把
            // 具体问题可以看这个帖子：http://social.msdn.microsoft.co
            Process currentproc = Process.GetCurrentProcess();
            Process[] processcollection = Process.GetProcessesByNa
           //  该程序已经运行,
            if (processcollection.Length >= 1)
            {
                foreach (Process process in processcollection)
                {
```

```
                        if (process.Id != currentproc.Id)
                        {
                            // 如果进程的句柄为0，即代表没有找到该窗体，即该窗
                            if (process.MainWindowHandle.ToInt32() == (
                            {
                                // 获得窗体句柄
                                formhwnd = FindWindow(null, "Form1");
                                // 重新显示该窗体并切换到带入到前台
                                ShowWindow(formhwnd, SW_RESTORE);
                                SwitchToThisWindow(formhwnd, true);
                            }
                            else
                            {
                                // 如果窗体没有隐藏，就直接切换到该窗体并带入到
                                // 因为窗体除了隐藏到托盘，还可以最小化
                                SwitchToThisWindow(process.MainWindowHa
                            }
                        }
                    }
                }
                else
                {
                    Application.EnableVisualStyles();
                    Application.SetCompatibleTextRenderingDefault(false
                    Application.Run(new Form1());
                }
                #endregion
            }
        }
    }
```

四、程序实现效果

四种实现方式的运行效果都是差不多的，这里就以实现方式一作为演示的，具体实现效果如下图：



五、总结

写这个专题主要是看到原因是看到论坛中有些朋友问了这样的问题，并且本人也回答了，所以就总结下具体的实现代码来帮助遇到同样问题的朋友做一个参考，同时也是对自己一个学习的积累和复习。下面附上程序所有源码：

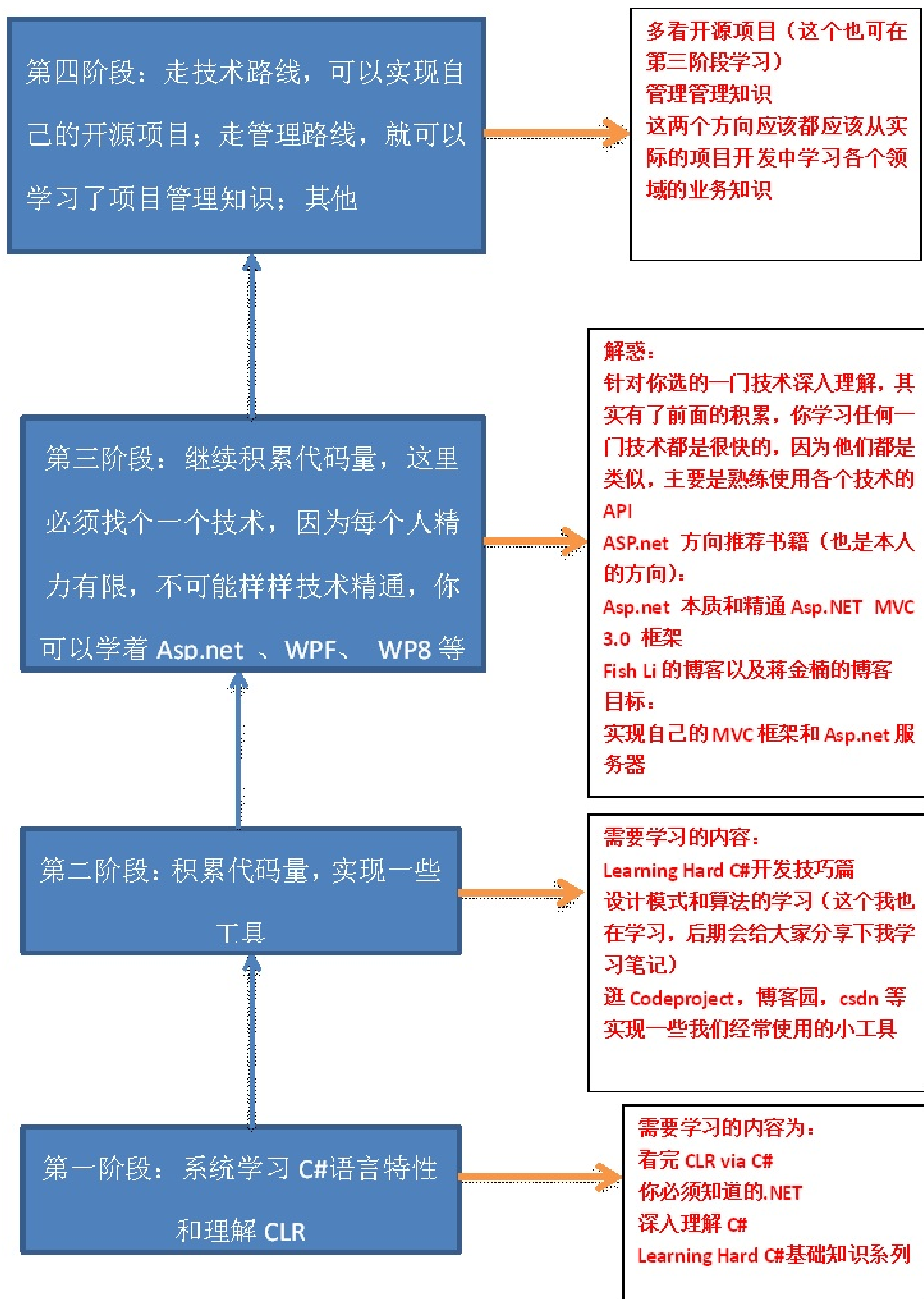
本专题程序源码：http://files.cnblogs.com/zhili/OnlyInstanceRunning.zip

[C# 开发技巧]实现属于自己的截图工具

一、引言

之前一直都是写一些C#基础知识的内容的，然而有些初学者可能看完了这些基础知识之后，会有这样一个疑惑的——我了解了这些基础知识之后，我想做一些工具怎么还是不会做的呢？那些基础知识到底有什么用的了？然而我刚开始写这个系列的初衷主要是我想系统地去研究下C#各个阶段的特性的，及时有些特性我知道它是怎么用的，但是每次遇到问题的时候确实百度可以可以解决很多问题，但是自己总是觉得有点“虚”，然而通过写完这个系列之后，我很多知识点都可以串起来了，可以做到一个举一反三的一个效果的，当我遇到实际问题的也不可能完全自己写出来，同样也会百度找解决方案，但是此时我却没有 “虚”的感觉，因为我知道这个东西，并且我也知道如何正确的百度这个问题。所以对于基础知识的学习还是很有必要的，因为系统学完之后，你可以更好地找到你遇到问题的答案，因为我有时候会看到一些朋友在QQ群中提到，遇到某个问题都不知道百度什么的，然而系统地学习基础完全可以帮助你快速地百度，（其实找答案也是一种能力），然而对于第一个疑惑的解答就是——系统学习完，确实刚开始的确开发工具不会做，但是实际写代码是很简单，并且现在大部分应用你百度下都可以找到的，所以代码并不是问题，主要是解决问题的思路，并且实际工具的开发也是对一个基础知识的巩固，从而对问题达到一个举一反三的效果。

上面说了这么多的（可能说的有点多），主要是让大家明白，系统学习**C#**基础知识是很有必要的，系统学习完**C#**基础知识之后就是代码量的积累了，也就是自己做一些小工具，积累到一定代码量之后，就可以尝试写写一些大的项目或开源项目等，所以在后面的系列中将会分享一些具体工具的开发，同时这也是我自己的一个学习的计划，这里分享给大家希望对一些迷茫的朋友有所帮助。如果你现在还没有明确或更好地目标，并且也是从事**.NET**工作或学习的朋友，那就和我一起静下心来学编程，下面是我的一个学习方向图（可能多少有点偏差，相信大致意图大家可以明白）：



二、 实现思路

啰嗦了这么多，下面就具体介绍下实现截图工具的实现思路。

为了让大家更清楚地知道如何去实现自己的截图工具，首先我来描述下截图的一个过程——我们使用QQ的截图工具和Windows 自带的截图工具都可以发现，当我们点击QQ窗体中的截图按钮时，此时我们将看到一个全屏图片，然后我们可以在其上截图,当鼠标左键按下时，即代表开始截图，并我们可以移动鼠标来改变截图的大小，鼠标弹起时即代表结束截图，此时我们可以双击矩形区域完全截图，并且可以通过粘贴操作把截取的图片粘贴到聊天窗口的发送区，鼠标右键点击则是退出截图。这样我们截图的过程描述就是这样的，从这个描述中我们就可以抽象出实现我们截图工具的思路来：

1. 从 “此时我们将看到一个全屏图片”这句话描述我们应该抽象为——对于QQ截图工具的实现来说，我们看到的这个全屏图片其实并不是一张“图片”(这里最好不要钻空子)，而是一个窗体，这个窗体我们命名为 “截图窗体”，只是把窗体的背景图片设置为全屏图片。说到这里，一些没有研究过QQ截图工具的人开始有疑问了——我们看到的是窗体？那为什么边框的，即没有最大化按钮，最下化按钮的呢？（对于这点的解释就是，程序中可以设置Form的BorderStyle属性为none的方式来隐藏掉边框）。
2. 既然要设置窗体的背景图片为全屏图片，我们知道设置背景图片只需要设置窗体的BackgroundImage.aspx)属性就好了，但是全屏图片怎么获取呢？既然是全屏图片，自然我就应该使窗体最大化话了，不然我们看到只是一个没有边框的“小图片”了，而不是一个全屏的图片。下面是具体实现这个分析的代码：

```
**// 通过Graphics的CopyFromScreen方法把全屏图片的拷贝到我们定义好的一个和
**// 拷贝完成之后，CatchBmp就是全屏图片的拷贝了，然后指定为截图窗体背景图片
            // 新建一个和屏幕大小相同的图片
            Bitmap CatchBmp = new Bitmap(Screen.AllScreens[0].Bounc

            // 创建一个画板，让我们可以在画板上画图
            // 这个画板也就是和屏幕大小一样大的图片
            // 我们可以通过Graphics这个类在这个空白图片上画图
            Graphics g = Graphics.FromImage(CatchBmp);

            // 把屏幕图片拷贝到我们创建的空白图片 CatchBmp中
            g.CopyFromScreen(new Point(0, 0), new Point(0, 0), new

            // 创建截图窗体
            cutter = new Cutter();

            // 指示窗体的背景图片为屏幕图片
            cutter.BackgroundImage = CatchBmp;
```

3. 从 “然后我们可以在其上截图”这句话中我们抽象为——其实我们截图操作，从程序角度来说就是我们在这个最大化的窗体中画图，可能这个对一些不了解GDI+画图的朋友有些难理解，这里做个比喻——我们会拿笔在纸上画图，我们可以用比画三角形，矩形已经各种图形，此时纸就是我们一个画板，笔是用来画图图形的，同时笔也是有颜色和粗细的，我们可以用红色水笔画，画出来的图就是红色的了，也可以用黑色水笔画，自然画出来的就是黑色的了，同样，在**GDI+**也就是**Graphics Device Interface Plus**也就是图形设备接口，在**.NET** 中也提供了一些这样的类来让我们实现对图像的访问，也就是我们可以使用**.NET**中提供的类来进行 **“画画”**，

要画画当然必须要有画板吧（我们开始比喻中纸就是画板）,在.NET 类中Graphics类就是对画板的抽象，画板可以由三种方式创建：（1）从图片或继承自图像对象中创建；（2）从窗体或控件的Paint事件中创建；（3）利用窗体或控件的CreateGraphics方法创建。有了画板之后，当然就需要笔来画画了，在.NET 中Pen类就是起到笔的作用，在构造函数中可以指定笔的颜色和粗细，有了笔之后就是开始画图了，在.NET中也同样提供了一些方法来完成画图，如DrawRectangle方法——画矩形

4. 从 “当鼠标左键按下时，即代表开始截图，并我们可以移动鼠标来改变截图的大小，鼠标弹起时即代表结束截图，此时我们可以双击矩形区域完全截图，并且可以通过粘贴操作把截取的图片粘贴到聊天窗口的发送区，鼠标右键点击则是退出截图”这些描述中可以抽象为——鼠标的移动，按下，弹起等操作，在程序角度来说，也就是实现截图窗体的MouseMove事件（对应于鼠标移动），MouseDown事件（对应于鼠标左键按下），MouseClick事件（对应于鼠标右键结束截图）、MouseUp(对应于鼠标弹起结束截图)和MouseDoubleClick(鼠标双击矩形区域完全截图，并可以通过粘贴操作把截取的图片粘贴到聊天窗口的发送区，既然可以进行粘贴操作来获得截取图片，所以必须在该事件中对剪切板设置截图图片)，3和4的分析过程也是截图功能的核心实现，对应于下面的代码（代码中有详细解释，并且大家理解的时候可以结合3和4的分析）：

```
/// <summary>
        /// 鼠标右键点击结束截图
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void Cutter_MouseClick(object sender, MouseEventArg
        {
            if (e.Button == MouseButtons.Right)
            {
                this.DialogResult = DialogResult.OK;
                this.Close();
            }
        }

        /// <summary>
        /// 鼠标按下事件处理程序
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void Cutter_MouseDown(object sender, MouseEventArgs
        {
            // 鼠标左键按下是开始画图，也就是截图
            if (e.Button == MouseButtons.Left)
            {
                // 如果捕捉没有开始
                if (!CatchStart)
                {
                    CatchStart = true;
                    // 保存此时鼠标按下坐标
                    DownPoint = new Point(e.X, e.Y);
                }
```

```
            }
        }

        /// <summary>
        /// 鼠标移动事件处理程序，即用户改变截图大小的处理
        ///   这个方法是截图功能的核心方法，也就是绘制截图
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void Cutter_MouseMove(object sender, MouseEventArgs
        {
            // 确保截图开始
            if (CatchStart)
            {
                // 新建一个图片对象，让它与屏幕图片相同
                Bitmap copyBmp = (Bitmap)originBmp.Clone();

                // 获取鼠标按下的坐标
                Point newPoint = new Point(DownPoint.X, DownPoint.Y

                // 新建画板和画笔
                Graphics g = Graphics.FromImage(copyBmp);
                Pen p = new Pen(Color.Red, 1);

                // 获取矩形的长宽
                int width = Math.Abs(e.X - DownPoint.X);
                int height = Math.Abs(e.Y-DownPoint.Y);
                if (e.X < DownPoint.X)
                {
                    newPoint.X = e.X;
                }
                if (e.Y < DownPoint.Y)
                {
                    newPoint.Y = e.Y;
                }

                CatchRectangle = new Rectangle(newPoint, new Size(w

                // 将矩形画在画板上
                g.DrawRectangle(p, CatchRectangle);

                // 释放目前的画板
                g.Dispose();
                p.Dispose();
                // 从当前窗体创建新的画板
                Graphics g1 = this.CreateGraphics();

                // 将刚才所画的图片画到截图窗体上
                // 为什么不直接在当前窗体画图呢？
                // 如果自己解决将矩形画在窗体上，会造成图片抖动并且有无数个矩
                // 这样实现也属于二次缓冲技术
                g1.DrawImage(copyBmp, new Point(0, 0));
                g1.Dispose();
```

```
                // 释放拷贝图片，防止内存被大量消耗
                copyBmp.Dispose();
            }
        }

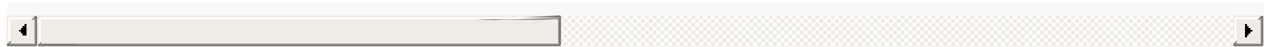
        /// <summary>
        /// 鼠标左键弹起事件
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void Cutter_MouseUp(object sender, MouseEventArgs e
        {
          if (e.Button == MouseButtons.Left)
            {
                // 如果截图已经开始，鼠标左键弹起设置截图完成
                if (CatchStart)
                {
                    CatchStart = false;
                    CatchFinished = true;
                }
            }
        }

        /// <summary>
        /// 鼠标双击事件，如果鼠标位于矩形内，则将矩形内的图片保存到剪切板中
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void Cutter_MouseDoubleClick(object sender, MouseEv
        {
            if (e.Button == MouseButtons.Left && CatchFinished)
            {
                // 新建一个与矩形一样大小的空白图片
                Bitmap CatchedBmp = new Bitmap(CatchRectangle.Width

                Graphics g = Graphics.FromImage(CatchedBmp);

                // 把originBmp中指定部分按照指定大小画到空白图片上
                // CatchRectangle指定originBmp中指定部分
                // 第二个参数指定绘制到空白图片的位置和大小
                // 画完后CatchedBmp不再是空白图片了，而是具有与截取的图片-
                g.DrawImage(originBmp, new Rectangle(0, 0, CatchRec

                // 将图片保存到剪切板中
                Clipboard.SetImage(CatchedBmp);
                g.Dispose();
                CatchFinished = false;
                this.BackgroundImage = originBmp;
                CatchedBmp.Dispose();
                this.DialogResult = DialogResult.OK;
                this.Close();
            }
        }
```



View Code

5 到第4点为止，截图的功能已经分析完了，之后就是当我们使用QQ截图的时候，我们除了可以点击聊天窗口中的截图按钮来进行截图外，还可以通过按下Alt+Ctrl+A来进行截图，要实现这个功能的思路也很简单——即当聊天窗体加载的时候对热键（程序中我定义的热键是“Alt+Ctrl+C”）进行注册（此时调用了Win32中**RegisterHotKey**方法来完成热键的注册），当聊天窗体关闭时进行对热键的卸载，防止对热键进行多次注册，此时调用Win32中的**UnregisterHotKey**方法来完成，具体的实现代码为：

```
/// <summary>
        /// 窗体加载事件处理
        /// 在窗体加载时注册热键
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void 聊天窗体_Load(object sender, EventArgs e)
        {
            uint ctrlHotKey = (uint)(KeyModifiers.Alt|KeyModifiers.
            // 注册热键为Alt+Ctrl+C, "100"为唯一标识热键
            HotKey.RegisterHotKey(Handle, 100, ctrlHotKey, Keys.C);
        }

        /// <summary>
        /// 窗体关闭时处理程序
        /// 窗体关闭时取消热键注册
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void 聊天窗体_FormClosing(object sender, FormClosing
        {
            // 卸载热键
            HotKey.UnregisterHotKey(Handle, 100);
        }

        #endregion

        // 热键按下执行的方法
        private void GlobalKeyProcess()
        {
            this.WindowState = FormWindowState.Minimized;
            // 窗口最小化也需要一定时间
            Thread.Sleep(200);
            btnCutter.PerformClick();
        }

        /// <summary>
        /// 重写WndProc()方法，通过监视系统消息，来调用过程
        /// 监视Windows消息
```

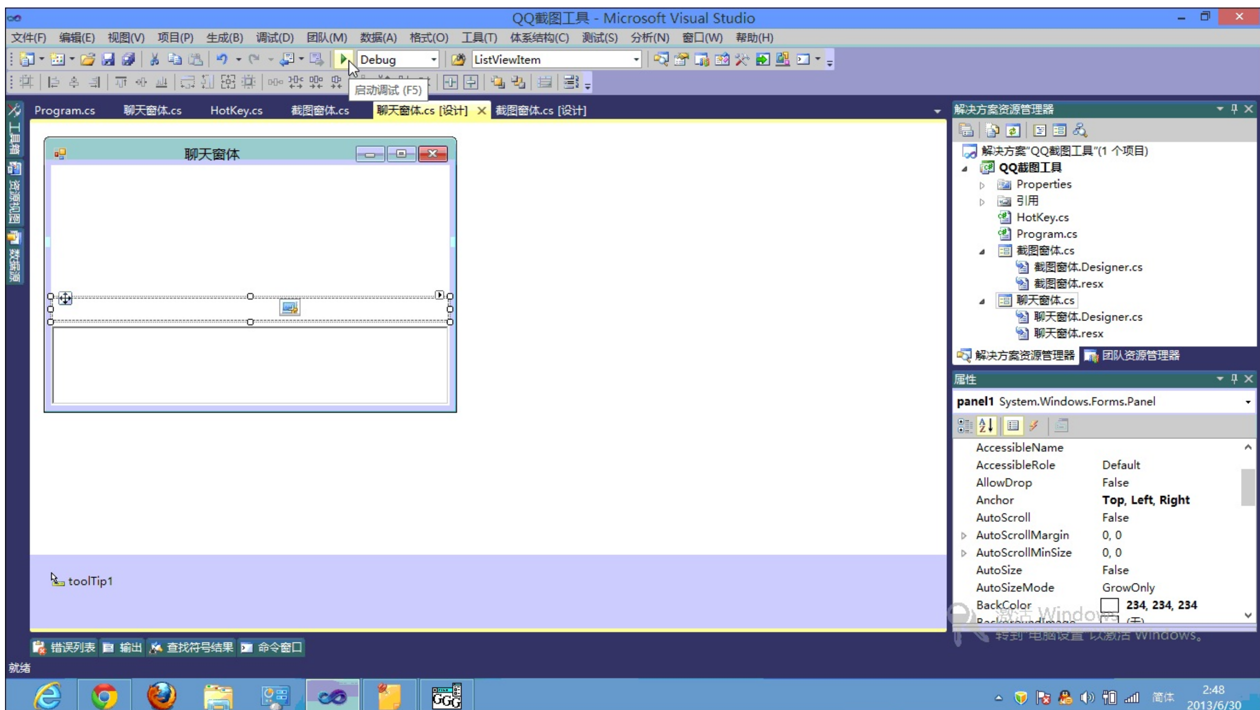
```
        /// </summary>
        /// <param name="m"></param>
        protected override void WndProc(ref Message m)
        {
            //如果m.Msg的值为0x0312那么表示用户按下了热键
            const int WM_HOTKEY = 0x0312;
            switch (m.Msg)
            {
                case WM_HOTKEY:
                    if (m.WParam.ToString() == "100")
                    {
                        GlobalKeyProcess();
                    }

                    break;
            }

            // 将系统消息传递自父类的WndProc
            base.WndProc(ref m);
        }
```

三、实现效果

上面已经介绍了实现QQ截图的一个思路的，朋友们是不是迫不及待想看看该程序的一个效果了？下面就通过一个动画来让大家更形象地看到程序的运行效果的：



四、总结

到这里QQ截图的介绍部分就到这里了，本工具的实现自认为讲解的非常通俗易懂的，希望大家可以这样觉得并且可以更清晰地明白QQ截图的实现思路的，下面附上本专题的所有源码和一个高仿QQ截图的文章：

本专题源码：http://files.cnblogs.com/zhili/QQ%E6%88%AA%E5%9B%BE%E5%B7%A5%E5%85%B7.zip

高仿腾讯QQ实现：http://blog.csdn.net/crystal_lz/article/details/8274277

[C# 开发技巧]如何使不符合要求的元素等于离它最近的一个元素

一、问题描述

今天在MSDN论坛中看到这样的一个问题，觉得非常锻炼思维能力，所以这里记录下来作为备份，题目的要求是这样的：

假设有一组字符串数组{"0","0","1","2","3",“0”,"4","0","0"}，如何查找使0等于离它最近的且不为0的元素，如果离它最近的不为0的元素有两个，则等于上一个元素，即想得到重新赋值后这样的数组{"1","1","1","2","3",“3”,"4","4","4"}

二、实现思路

这里的实现思路摘自论坛中 zjyh26的回复，实现思路为：

- 1. 首先对数组里面的数字进行一次遍历，如果当前的值不为“0”把值添加进的结果数组中，否则对它进行处理。
- 2. 处理不为“0”的值的时候，用一种“等距离比较”的方法，找出等距离内的左右2个值，优先看左边的值是否为“0”，如果是的话跳过，如果不是的话将结果数组内的当前值替换为此值。
- 3. 距离（就是代码里面的j）的最大值为数组长度减去1，遍历的时候注意i-j的值不小于0，i+j的值要小于数组长度。

具体实现代码为：

```
string[] s = new String[9] { "0", "0", "1", "0", "0", "0", "4", '
            string[] result = new string[9];
            for (int i = 0; i < s.Length; i++)
            {

                if (s[i] != "0")
                {
                    result[i] = s[i];
                    continue;
                }

                // j是距离，初始化距离为1
                for (int j = 1; j < s.Length; j++)
                {
                    if (i - j >= 0)
                    {
                        // 左边距离为j的元素不等于0时
                        if (s[i - j] != "0")
                        {
                            result[i] = s[i - j];
                            break;
                        }
                    }
                    if (i + j < s.Length)
                    {
                        // 右边距离为j的元素不等于0时
                        if (s[i + j] != "0")
                        {
                            result[i] = s[i + j];
                            break;
                        }
                    }
                }
            }

            for (int i = 0; i < result.Length; i++)
            {
                Console.WriteLine(result[i]);
            }

            Console.ReadLine();
```

三、运行结果

file:///E:/学习/博客园例子/论坛问题测试/论坛问题测试/bin/Debug/论坛问题测试.EXE

```
1
1
1
1
1
4
4
4
3
```

C# 线程处理系列

[C# 线程处理系列]专题一：线程基础

引言：

最近一段时间都在研究关于.Net线程的内容，觉得线程是每个程序员都应该掌握的，所以写下这个线程的系列希望能给大家学习过程中一些帮助，同时也是自己对线程的巩固，当中如果有什么错漏还请大家指出，这样我们可以互相得到进步。

目录：

一、线程的介绍

二、线程调度和优先级

三、前台线程和后台线程

四、简单线程的使用

一、线程的介绍

在介绍线程之前， 很有必要知道什么是进程，以及与线程的关系。

进程(Process)是应用程序的实例要使用的资源的一个集合（从可以简化理解：进程就是一种资源，是应用程序所用的资源）。每个应用程序都在各自的进程中运行来确保应用程序不受其他应用程序的影响，如果一个应用程序失败了， 只会影响自己的进程，其他进程中的应用程序可以继续运行。进程是操作系统为我们提供的一种保护应用程序的一种机制。

线程是进程中基本执行单元， 一个进程中可以包含多个线程，在进程入口执行的第一个线程是一个进程的主线程，在.Net应用程序中，都是以Main()方法作为程序的入口的， 所以在程序运行过程中调用这个方法时，系统就会自动创建一个主线程。（他们之间的关系简单说：线程是进程的执行单元，进程是线程的一个容器了）。

二、线程调度和优先级

Windows之所以被称为抢占式多线程操作系统，是因为线程可以在任意时间被抢占，并调度另一个线程。每个线程都分配了从0~31的一个优先级。系统首先把高优先级的线程分配给CPU执行。Windows 支持7个相对线程优先级：Idle,Lowest,Below Normal,Normal,Above Normal,Highest和Time-Critical,Normal是默认的线程优先级，然而在程序中可以通过设置Thread的Priority属性来改变线程的优先级，它的类型为ThreadPriority枚举类型，包含枚举有：Lowest,BelowNormal,Normal,AboveNormal和Highest,CLR为自己保留了 Idle和Time-Critical优先级。具体每个枚举值含义如下表：

成员名称	说明
Lowest	Thread can be scheduled after threads with any other priority." data-guid="3e53547e0e9a509aff4a76382e494083">可以将 Thread何其他优先级的线程之后。
BelowNormal	Thread can be scheduled after threads with Normal priority and before those with Lowest priority." data-guid="f979b4a5dfbb5a35942312092b2cd47a">可以将 Thread Normal 优先级的线程之后，在具有 Lowest 优先级的线程之前。
Normal	Thread can be scheduled after threads with AboveNormal priority and before those with BelowNormal priority." data-guid="8b94d9644aacd17640f4971fc6e79dc7">可以将 Thread AboveNormal 优先级的线程之后，在具有 BelowNormal 优先级的线程之前。 Normal priority by default." data-guid="6fa67ded483ad3d242ea365f0ac225c4">默认情况下，线程具有 Normal 优先级。
AboveNormal	Thread can be scheduled after threads with Highest priority and before those with Normal priority." data-guid="d067edb8ea0b5bfd51a2591334b86fd7">可以将 Thread Highest 优先级的线程之后，在具有 Normal 优先级的线程之前。
Highest	Thread can be scheduled before threads with any other priority." data-guid="fc7a0e931c772d08fc98b6de2c82ab3c">可以将 Thread 其他优先级的线程之前。

三、前台线程和后台线程

在.net中线程分为前台线程和后台线程，在一个进程中，当所有前台线程停止运行时，CLR会强制结束仍在运行的任何后台线程，这些后台线程直接被终止，不会抛出异常。

所以我们应该在前台线程中执行我们确实要完成的事情，另外， 应该把非关键的任务使用后台线程，我们用Thread创建的是线程为前台线程。让我们通过下面的一段代码来看看前台线程和后台线成的区别：

```
using System;
using System.Threading;

 class Program
    {
        static void Main(string[] args)
        {
  // 创建一个新线程（默认为前台线程）
            Thread backthread = new Thread(Worker);

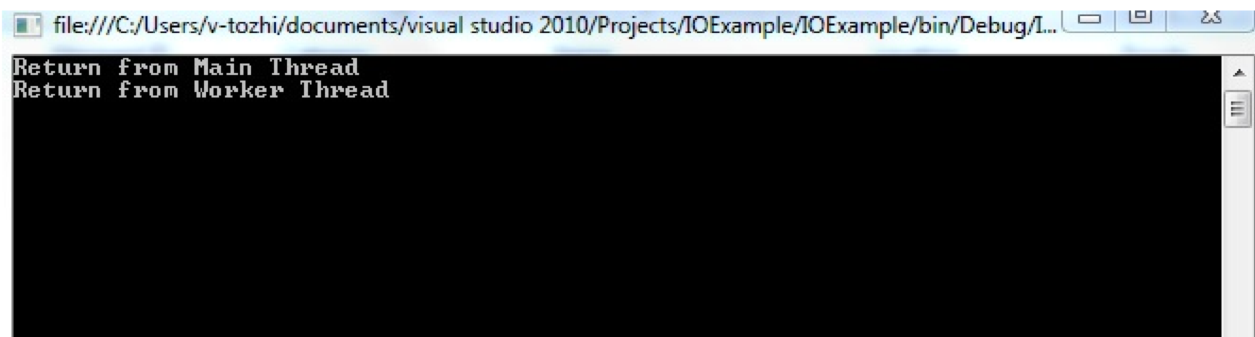
            // 使线程成为一个后台线程
            backthread.IsBackground = true;

            // 通过Start方法启动线程
            backthread.Start();

            // 如果backthread是前台线程，则应用程序大约5秒后才终止
            // 如果backthread是后台线程，则应用程序立即终止
            Console.WriteLine("Return from Main Thread");
        }
        private static void Worker()
        {
            // 模拟做10秒
            Thread.Sleep(5000);

            // 下面语句，只有由一个前台线程执行时，才会显示出来
            Console.WriteLine("Return from Worker Thread");
        }
}
```

运行上面代码可以发现：控制台中显示字符串: Return form Main Thread 后就退出了， 字符串 Return from Worker Thread字符串根本就没有显示，这是因为此时的backthread线程为后台线程，当主线程（执行Main方法的线程，主线程当然也是前台线程了）结束运行后，CLR会强制终止后台线程的运行，整个进程就被销毁了，并不会等待后台线程运行完后才销毁。如果把 backthread.IsBackground = true; 注释掉后， 就可以看到控制台过5秒后就输出 Return from Worker Thread。再在Worker方法最后加一句 代码：Console.Read(); 就可以看到这样的结果了：



注意：有些人可能会问我不想把 backthread.IsBackground = true;注释掉， 又想把 Worker()方法中的字符串输出在控制台上怎么做呢？ 其实是有解决的办法的， 我们可以调用thread.Join()方法来实现，Join()方法能保证主线程（前台线程）在异步线程thread（后台线程）运行结束后才会运行。

实现代码如下：

```
using System;
using System.Threading;
 class Program
    {
        static void Main(string[] args)
        {
 // 创建一个新线程（默认为前台线程）
            Thread backthread = new Thread(Worker);

            // 使线程成为一个后台线程
            backthread.IsBackground = true;

            // 通过Start方法启动线程
            backthread.Start();
            backthread.Join();

            // 模拟主线程的输出
            Thread.Sleep(2000);

            Console.WriteLine("Return from Main Thread");
            Console.Read();
        }

        private static void Worker()
        {
            // 模拟做3秒
            Thread.Sleep(3000);

            // 下面语句，只有由一个前台线程执行时，才会显示出来
            Console.WriteLine("Return from Worker Thread");

        }
}
```

运行结果（调用Join方法后后台线程会阻塞主线程所以主线程会后输出）：

```
file:///d:/my documents/visual studio 2010/Projects/Thread/Thread/bin...
Return from Worker Thread
Return from Main Thread
```

四、 简单线程的使用

其实在上面介绍前台线程和后台线程的时候已经通过ThreadStart委托创建了一个线程了，此时已经实现了一个多线程的一个过程，为此系列中将多线程也是做一个铺垫吧。下面通过**ParameterizedThreadStart**委托的方式来实现多线程。

以**ParameterizedThreadStart**委托的方式来实现多线程：

```
using System;
using System.Threading;
 class Program
    {
        static void Main(string[] args)
        {
// 创建一个新线程（默认为前台线程）
            Thread backthread = new Thread(new ParameterizedThreadS

            // 通过Start方法启动线程
            backthread.Start("123");

            // 如果backthread是前台线程，则应用程序大约5秒后才终止
            // 如果backthread是后台线程，则应用程序立即终止
            Console.WriteLine("Return from Main Thread");
        }

        private static void Worker(object data)
        {
            // 模拟做5秒
            Thread.Sleep(5000);

            // 下面语句，只有由一个前台线程执行时，才会显示出来
            Console.WriteLine(data + " Return from Worker Thread")
            Console.Read();
        }
}
```

注意：此时Worker方法传入了一个参数，并且Start方法也传递了一个字符传参数。对比与之前创建Thread的不同，

运行结果为：

```
file:///C:/Users/v-tozhi/documents/visual studio 2010/Projects/IOExample/IOExample/bin/Debug/I...
eturn from Main Thread
23 Return from Worker Thread
```

写到这里，本系列的第一篇差不多讲完了，在后续的文章将会介绍Thread方法的使用以及通过一些例子来展示他们的不同之处（像Abort()方法Interrupt方法等）对于线程的一些高级使用（如线程池，并行编程和PLINQ、线程同步和计时器）都会

在后续中讲到。希望本系列可以给初学线程的人有所帮助。

[C# 线程处理系列]专题二：线程池中的工作者线程

目录：

一、上节补充

对于Thread类还有几个常用方法需要说明的。

1.1 Suspend和Resume方法

这两个方法在.net Framework 1.0的时候就支持的方法，他们分别可以挂起线程和恢复挂起的线程。但在.net Framework 2.0以后的版本中这两个方法都过时了，MSDN的解释是这样：

警告：

Suspend and Resume methods to synchronize the activities of threads." data-guid="ff8e76fddd4f9d11989c7d25f5e342d1">不要使用 Suspend 和 Resume 方法来同步线程的活动。您无法知道挂起线程时它正在执行什么代码。AppDomain might be blocked." data-guid="7c4587074f4e52d11eadc59e58a09a57">如果您在安全权限评估期间挂起持有锁的线程，则 AppDomain中的其他线程可能被阻止。AppDomain that attempt to use that class are blocked." data-guid="f164549efd95a0a94479d4d1c0d1ceec">如果您在线程正在执行类构造函数时挂起它，则 AppDomain中尝试使用该类的其他线程将被阻止。这样很容易发生死锁。

对于这个解释可能有点抽象吧，让我们来看看一段代码可能会清晰点：

```
class Program
    {
        static void Main(string[] args)
        {
            // 创建一个线程来测试
            Thread thread1 = new Thread(TestMethod);
            thread1.Name = "Thread1";
            thread1.Start();
            Thread.Sleep(2000);
            Console.WriteLine("Main Thread is running");
            ////int b = 0;
            ////int a = 3 / b;
            ////Console.WriteLine(a);
            thread1.Resume();
            Console.Read();
        }

        private static void TestMethod()
        {
            Console.WriteLine("Thread: {0} has been suspended!", Th

            //将当前线程挂起
            Thread.CurrentThread.Suspend();
            Console.WriteLine("Thread: {0} has been resumed!", Thre
        }
    }
```

在上面这段代码中thread1线程是在主线程中恢复的，但当主线程发生异常时，这时候就thread1一直处于挂起状态，此时thread1所使用的资源就不能释放（除非强制终止进程），当另外线程需要使用这快资源的时候，这时候就很可能发生死锁现象。

上面一段代码还存在一个隐患，请看下面一小段代码：

```
class Program
    {
        static void Main(string[] args)
        {
            // 创建一个线程来测试
            Thread thread1 = new Thread(TestMethod);
            thread1.Name = "Thread1";
            thread1.Start();
            Console.WriteLine("Main Thread is running");
            thread1.Resume();
            Console.Read();
        }

        private static void TestMethod()
        {
            Console.WriteLine("Thread: {0} has been suspended!", Th
            Thread.Sleep(1000);

            //将当前线程挂起
            Thread.CurrentThread.Suspend();
            Console.WriteLine("Thread: {0} has been resumed!", Thre
        }
    }
```

当主线程跑（运行）的太快,做完自己的事情去唤醒thread1时，此时thread1还没有挂起而起唤醒thread1,此时就会出现异常了。并且上面使用的Suspend和Resume方法，编译器已经出现警告了，提示这两个方法已经过时， 所以在我们平时使用中应该尽量避免。

1.2 Abort和 Interrupt方法

Abort方法和Interrupt都是用来终止线程的，但是两者还是有区别的。

1、他们抛出的异常不一样，Abort 方法抛出的异常是ThreadAbortException，Interrupt抛出的异常为ThreadInterruptedException

2、调用interrupt方法的线程之后可以被唤醒，然而调用Abort方法的线程就直接被终止不能被唤醒的。

下面一段代码是掩饰Abort方法的使用

```
using System;
using System.Threading;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
```

```
            Thread abortThread = new Thread(AbortMethod);
            abortThread.Name = "Abort Thread";
            abortThread.Start();
            Thread.Sleep(1000);
            try
            {
                abortThread.Abort();
            }
            catch
            {
                Console.WriteLine("{0} Exception happen in Main Thr
                Console.WriteLine("{0} Status is:{1} In Main Threac
            }
            finally
            {
                Console.WriteLine("{0} Status is:{1} In Main Threac
            }

            abortThread.Join();
            Console.WriteLine("{0} Status is:{1} ", abortThread.Nar
            Console.Read();

        }

        private static void AbortMethod()
        {
            try
            {
                Thread.Sleep(5000);
            }
            catch(Exception e)
            {
                Console.WriteLine(e.GetType().Name);
                Console.WriteLine("{0} Exception happen In Abort Tł
                Console.WriteLine("{0} Status is:{1} In Abort Threa
            }
            finally
            {
                Console.WriteLine("{0} Status is:{1} In Abort Threa
            }
        }
    }
```

运行结果：

```
file:///C:/Users/v-tozhi/documents/visual studio 2010/Projects/OfficeGetStarted/ConsoleApplicati...
ThreadAbortException
Abort Thread Exception happen In Abort Thread
Abort Thread Status is:AbortRequested In Abort Thread
Abort Thread Status is:AbortRequested In Abort Thread
Abort Thread Status is:AbortRequested In Main Thread
Abort Thread Status is:Aborted
```

从运行结果可以看出，调用Abort方法的线程引发的异常类型为ThreadAbortException, 以及异常只会在 调用Abort方法的线程中发生，而不会在主线程中抛出，并且调用Abort方法后线程的状态不是立即改变为Aborted状态，而是从AbortRequested->Aborted。

Interrupt方法：

```
using System;
using System.Threading;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        { Thread interruptThread = new Thread(AbortMethod);
            interruptThread.Name = "Interrupt Thread";
            interruptThread.Start();
            interruptThread.Interrupt();

            interruptThread.Join();
            Console.WriteLine("{0} Status is:{1} ", interruptThread
            Console.Read();
        }

        private static void AbortMethod()
        {
            try
            {
                Thread.Sleep(5000);
            }
            catch(Exception e)
            {
                Console.WriteLine(e.GetType().Name);
                Console.WriteLine("{0} Exception happen In Interru
                Console.WriteLine("{0} Status is:{1} In Interrupt
            }
            finally
            {
                Console.WriteLine("{0} Status is:{1} In Interrupt
            }

        }
    }
}
```

运行结果：

file:///C:/Users/v-tozhi/documents/visual studio 2010/Projects/OfficeGetStarted/ConsoleApplicati...

```
ThreadInterruptedException
Interrupt Thread Exception happen In Interrupt Thread
Interrupt Thread Status is:Running In Interrupt Thread
Interrupt Thread Status is:Running In Interrupt Thread
Interrupt Thread Status is:Stopped
```

从结果中可以得到，调用Interrupt方法抛出的异常为：ThreadInterruptException, 以及当调用Interrupt方法后线程的状态应该是中断的， 但是从运行结果看此时的线程因为了Join,Sleep方法而唤醒了线程，为了进一步解释调用Interrupt方法的线程可以被唤醒， 我们可以在线程执行的方法中运用循环，如果线程可以唤醒，则输出结果中就一定会有循环的部分，然而调用Abort方法线程就直接终止，就不会有循环的部分，下面代码相信大家看后肯定会更加理解两个方法的区别的：

```
using System;
using System.Threading;

namespace ConsoleApplication2
{
    class Program
    {
        static void Main(string[] args)
        {
            Thread thread1 = new Thread(TestMethod);
            thread1.Start();
            Thread.Sleep(100);

            thread1.Interrupt();
            Thread.Sleep(3000);
            Console.WriteLine("after finnally block, the Thread1 st
            Console.Read();
        }
        private static void TestMethod()
        {

            for (int i = 0; i < 4; i++)
            {
                try
                {
                    Thread.Sleep(2000);
                    Console.WriteLine("Thread is Running");
                }
                catch (Exception e)
                {
                    if (e != null)
                    {
                        Console.WriteLine("Exception {0} throw ", e
                    }
                }
                finally
                {
                    Console.WriteLine("Current Thread status is:{0]
                }
            }
        }
    }
}
```

运行结果为：

file:///C:/Users/v-tozhi/documents/visual studio 2010/Projects/OfficeGetStarted/MultilThread/Con...

```
Exception ThreadInterruptedException throw
Current Thread status is:Running
Thread is Running
Current Thread status is:Running
after finnally block, the Thread1 status is:WaitSleepJoin
Thread is Running
Current Thread status is:Running
Thread is Running
Current Thread status is:Running
```

如果把上面的 thread1.Interrupt();改为 thread1.Abort(); 运行结果为：

file:///C:/Users/v-tozhi/documents/visual studio 2010/Projects/OfficeGetStarted/MultilThread/Con...

```
Exception ThreadAbortException throw
Current Thread status is:AbortRequested
after finnally block, the Thread1 status is:Aborted
```

二、线程池基础

首先，创建和销毁线程是一个要耗费大量时间的过程，另外，太多的线程也会浪费内存资源，所以通过Thread类来创建过多的线程反而有损于性能，为了改善这样的问题，.net中就引入了线程池。

线程池形象的表示就是存放应用程序中使用的线程的一个集合（就是放线程的地方，这样线程都放在一个地方就好管理了）。CLR初始化时，线程池中是没有线程的，在内部，线程池维护了一个操作请求队列，当应用程序想执行一个异步操作时，就调用一个方法，就将一个任务放到线程池的队列中，线程池中代码从队列中提取任务，将这个任务委派给一个线程池线程去执行，当线程池线程完成任务时，线程不会被销毁，而是返回到线程池中，等待响应另一个请求。由于线程不被销毁，这样就可以避免因为创建线程所产生的性能损失。

注意：通过线程池创建的线程默认为后台线程，优先级默认为**Normal.**

三、通过线程池的工作者线程实现异步

3.1 创建工作者线程的方法

public static bool QueueUserWorkItem (WaitCallback callBack);

public static bool QueueUserWorkItem(WaitCallback callback, Object state);

这两个方法向线程池的队列添加一个工作项（work item）以及一个可选的状态数据。然后，这两个方法就会立即返回。

工作项其实就是由callback参数标识的一个方法，该方法将由线程池线程执行。同时写的回调方法必须匹配System.Threading.WaitCallback委托类型，定义为：

public delegate void WaitCallback(Object state);

下面演示如何通过线程池线程来实现异步调用：

```
using System;
using System.Threading;

namespace ThreadPoolUse
{
    class Program
    {
        static void Main(string[] args)
        {
            // 设置线程池中处于活动的线程的最大数目
            // 设置线程池中工作者线程数量为1000，I/O线程数量为1000
            ThreadPool.SetMaxThreads(1000, 1000);
            Console.WriteLine("Main Thread: queue an asynchronous r
            PrintMessage("Main Thread Start");

            // 把工作项添加到队列中，此时线程池会用工作者线程去执行回调方法
            ThreadPool.QueueUserWorkItem(asyncMethod);
            Console.Read();
        }

        // 方法必须匹配WaitCallback委托
        private static void asyncMethod(object state)
        {
            Thread.Sleep(1000);
            PrintMessage("Asynchoronous Method");
            Console.WriteLine("Asynchoronous thread has worked ");
        }

        // 打印线程池信息
        private static void PrintMessage(String data)
        {
            int workthreadnumber;
            int iothreadnumber;

            // 获得线程池中可用的线程，把获得的可用工作者线程数量赋给workthr
            // 获得的可用I/O线程数量给iothreadnumber变量
            ThreadPool.GetAvailableThreads(out workthreadnumber, ou

            Console.WriteLine("{0}\n CurrentThreadId is {1}\n Curre
                data,
                Thread.CurrentThread.ManagedThreadId,
                Thread.CurrentThread.IsBackground.ToString(),
                workthreadnumber.ToString(),
                iothreadnumber.ToString());
        }
    }
}
```

运行结果：

```
file:///C:/Users/v-tozhi/documents/visual studio 2010/Projects/OfficeGetStarted/MultilThread/Thre...
Main Thread: queue an asynchronous method
Main Thread Start
 CurrentThreadId is 9
 CurrentThread is background :False
 WorkerThreadNumber is:1000
 IOThreadNumbers is: 1000

Asynchoronous Method
 CurrentThreadId is 10
 CurrentThread is background :True
 WorkerThreadNumber is:999
 IOThreadNumbers is: 1000

Asynchoronous thread has worked
```

从结果中可以看出，线程池中的可用的工作者线程少了一个，用去执行回调方法了。

ThreadPool.QueueUserWorkItem(WaitCallback callback,Object state) 方法可以把object对象作为参数传送到回调函数中,使用和ThreadPool.QueueUserWorkItem(WaitCallback callback)的使用和类似，这里就不列出了。

3.2 协作式取消

.net Framework提供了取消操作的模式， 这个模式是协作式的。为了取消一个操作，首先必须创建一个**System.Threading.CancellationTokenSource**对象。

下面代码演示了协作式取消的使用，主要实现当用户在控制台敲下回车键后就停止数数方法。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;

namespace ConsoleApplication3
{
    class Program
    {
        static void Main(string[] args)
        {
            ThreadPool.SetMaxThreads(1000, 1000);
            Console.WriteLine("Main thread run");
            PrintMessage("Start");
            Run();
            Console.ReadKey();
```

```
        }

        private static void Run()
        {
            CancellationTokenSource cts = new CancellationTokenSour

            // 这里用Lambda表达式的方式和使用委托的效果一样的，只是用了Lamb
            // 这在这里就是让大家明白怎么lambda表达式如何由委托转变的
            ////ThreadPool.QueueUserWorkItem(o => Count(cts.Token,

            ThreadPool.QueueUserWorkItem(callback, cts.Token);

            Console.WriteLine("Press Enter key to cancel the operat
            Console.ReadLine();

            // 传达取消请求
            cts.Cancel();
        }

        private static void callback(object state)
        {
            Thread.Sleep(1000);
            PrintMessage("Asynchoronous Method Start");
            CancellationToken token =(CancellationToken)state;
            Count(token, 1000);
        }

        // 执行的操作，当受到取消请求时停止数数
        private static void Count(CancellationToken token,int count
        {
            for (int i = 0; i < countto; i++)
            {
                if (token.IsCancellationRequested)
                {
                    Console.WriteLine("Count is canceled");
                    break;
                }

                Console.WriteLine(i);
                Thread.Sleep(300);
            }

            Console.WriteLine("Cout has done");
        }

        // 打印线程池信息
        private static void PrintMessage(String data)
        {
            int workthreadnumber;
            int iothreadnumber;

            // 获得线程池中可用的线程，把获得的可用工作者线程数量赋给workthr
            // 获得的可用I/O线程数量给iothreadnumber变量
```

```
                ThreadPool.GetAvailableThreads(out workthreadnumber, ou

                Console.WriteLine("{0}\n CurrentThreadId is {1}\n Curre
                    data,
                    Thread.CurrentThread.ManagedThreadId,
                    Thread.CurrentThread.IsBackground.ToString(),
                    workthreadnumber.ToString(),
                    iothreadnumber.ToString());
            }
        }
    }
```

运行结果：

```
file:///C:/Users/v-tozhi/documents/visual studio 2010/Projects/OfficeGetStarted/MultilThread/Con...
Main thread run
Start
 CurrentThreadId is 9
 CurrentThread is background :False
 WorkerThreadNumber is:1000
 IOThreadNumbers is: 1000

Press Enter key to cancel the operation

Asynchoronous Method Start
 CurrentThreadId is 10
 CurrentThread is background :True
 WorkerThreadNumber is:999
 IOThreadNumbers is: 1000

0
1
2
3
4
5
6
7
8
9
10

Count is canceled
Cout has done
```

四、使用委托实现异步

通过调用ThreadPool的QueueUserWorkItem方法来来启动工作者线程非常方便，但委托WaitCallback指向的是带有一个参数的无返回值的方法，如果我们实际操作中需要有返回值，或者需要带有多个参数， 这时通过这样的方式就难以实现， 为了解决这样的问题，我们可以通过委托来建立工作这线程，

下面代码演示了使用委托如何实现异步：

```
using System;
using System.Threading;

namespace Delegate
{
    class Program
```

```
    {
        // 使用委托的实现的方式是使用了异步变成模型APM（Asynchronous Progr
        // 自定义委托
        private delegate string MyTestdelegate();

        static void Main(string[] args)
        {
            ThreadPool.SetMaxThreads(1000, 1000);
            PrintMessage("Main Thread Start");

            //实例化委托
            MyTestdelegate testdelegate = new MyTestdelegate(asyncM

            // 异步调用委托
            IAsyncResult result = testdelegate.BeginInvoke(null, nu

            // 获取结果并打印出来
            string returndata = testdelegate.EndInvoke(result);
            Console.WriteLine(returndata);

            Console.ReadLine();
        }
        private static string asyncMethod()
        {
            Thread.Sleep(1000);
            PrintMessage("Asynchoronous Method");
            return "Method has completed";
        }

        // 打印线程池信息
        private static void PrintMessage(String data)
        {
            int workthreadnumber;
            int iothreadnumber;

            // 获得线程池中可用的线程，把获得的可用工作者线程数量赋给workthr
            // 获得的可用I/O线程数量给iothreadnumber变量
            ThreadPool.GetAvailableThreads(out workthreadnumber, ou

            Console.WriteLine("{0}\n CurrentThreadId is {1}\n Curre
                data,
                Thread.CurrentThread.ManagedThreadId,
                Thread.CurrentThread.IsBackground.ToString(),
                workthreadnumber.ToString(),
                iothreadnumber.ToString());
        }
    }
}
```

运行结果：

```
file:///C:/Users/v-tozhi/documents/visual studio 2010/Projects/OfficeGetStarted/MultiIThread/Del...
Main Thread Start
 CurrentThreadId is 10
 CurrentThread is background :False
 WorkerThreadNumber is:1000
 IOThreadNumbers is: 1000

Asynchoronous Method
 CurrentThreadId is 11
 CurrentThread is background :True
 WorkerThreadNumber is:999
 IOThreadNumbers is: 1000

Method has completed
```

五、任务

同样 任务的引入也是为了解决通过ThreadPool.QueueUserWorkItem中限制的问题，

下面代码演示通过任务来实现异步：

5.1 使用任务来实现异步

```
using System;
using System.Threading;
using System.Threading.Tasks;

namespace TaskUse
{
    class Program
    {
        static void Main(string[] args)
        {
            ThreadPool.SetMaxThreads(1000, 1000);
            PrintMessage("Main Thread Start");
            // 调用构造函数创建Task对象,
            Task<int> task = new Task<int>(n => asyncMethod((int)n)

            // 启动任务
            task.Start();
            // 等待任务完成
            task.Wait();
            Console.WriteLine("The Method result is: "+task.Result)

            Console.ReadLine();
        }

        private static int asyncMethod(int n)
        {
            Thread.Sleep(1000);
            PrintMessage("Asynchoronous Method");
```

```
            int sum = 0;
            for (int i = 1; i < n; i++)
            {
                // 如果n太大，使用checked使下面代码抛出异常
                checked
                {
                    sum += i;
                }
            }

            return sum;
        }

        // 打印线程池信息
        private static void PrintMessage(String data)
        {
            int workthreadnumber;
            int iothreadnumber;

            // 获得线程池中可用的线程，把获得的可用工作者线程数量赋给workthr
            // 获得的可用I/O线程数量给iothreadnumber变量
            ThreadPool.GetAvailableThreads(out workthreadnumber, ou

            Console.WriteLine("{0}\n CurrentThreadId is {1}\n Curre
                data,
                Thread.CurrentThread.ManagedThreadId,
                Thread.CurrentThread.IsBackground.ToString(),
                workthreadnumber.ToString(),
                iothreadnumber.ToString());
        }
    }
}
```

运行结果：

```
file:///C:/Users/v-tozhi/documents/visual studio 2010/Projects/OfficeGetStarted/MultilThread/Task...
Main Thread Start
 CurrentThreadId is 10
 CurrentThread is background :False
 WorkerThreadNumber is:1000
 IOThreadNumbers is: 1000

Asynchoronous Method
 CurrentThreadId is 7
 CurrentThread is background :True
 WorkerThreadNumber is:999
 IOThreadNumbers is: 1000

The Method result is: 45
```

5.2 取消任务

如果要取消任务， 同样可以使用一个CancellationTokenSource对象来取消一个Task.

下面代码演示了如何来取消一个任务：

```
using System;
using System.Threading;
using System.Threading.Tasks;

namespace TaskUse
{
    class Program
    {
        static void Main(string[] args)
        {
            ThreadPool.SetMaxThreads(1000, 1000);
            PrintMessage("Main Thread Start");
            CancellationTokenSource cts = new CancellationTokenSour

            // 调用构造函数创建Task对象,将一个CancellationToken传给Task
            Task<int> task = new Task<int>(n => asyncMethod(cts.Tol

            // 启动任务
            task.Start();

            // 延迟取消任务
            Thread.Sleep(3000);

            // 取消任务
            cts.Cancel();
            Console.WriteLine("The Method result is: " + task.Resul
            Console.ReadLine();
        }

        private static int asyncMethod(CancellationToken ct, int n)
        {
            Thread.Sleep(1000);
            PrintMessage("Asynchoronous Method");

            int sum = 0;
            try
            {
                for (int i = 1; i < n; i++)
                {
                    // 当CancellationTokenSource对象调用Cancel方法时,
                    // 就会引起OperationCanceledException异常
                    // 通过调用CancellationToken的ThrowIfCancellatio
                    // 这个方法和CancellationToken的IsCancellationReq
                    ct.ThrowIfCancellationRequested();
                    Thread.Sleep(500);
                    // 如果n太大，使用checked使下面代码抛出异常
                    checked
```

```
                    {
                        sum += i;
                    }
                }
            }
            catch (Exception e)
            {
                Console.WriteLine("Exception is:" + e.GetType().Nar
                Console.WriteLine("Operation is Canceled");
            }

            return sum;
        }

        // 打印线程池信息
        private static void PrintMessage(String data)
        {
            int workthreadnumber;
            int iothreadnumber;

            // 获得线程池中可用的线程，把获得的可用工作者线程数量赋给workthı
            // 获得的可用I/O线程数量给iothreadnumber变量
            ThreadPool.GetAvailableThreads(out workthreadnumber, ou

            Console.WriteLine("{0}\n CurrentThreadId is {1}\n Curre
                data,
                Thread.CurrentThread.ManagedThreadId,
                Thread.CurrentThread.IsBackground.ToString(),
                workthreadnumber.ToString(),
                iothreadnumber.ToString());
        }
    }
}
```

运行结果：

```
file:///C:/Users/v-tozhi/documents/visual studio 2010/Projects/OfficeGetStarted/MultilThread/Task...
Main Thread Start
 CurrentThreadId is 10
 CurrentThread is background :False
 WorkerThreadNumber is:1000
 IOThreadNumbers is: 1000

Asynchoronous Method
 CurrentThreadId is 11
 CurrentThread is background :True
 WorkerThreadNumber is:999
 IOThreadNumbers is: 1000

Exception is:OperationCanceledException
Operation is Canceled
The Method result is: 10
```

5.3 任务工厂

同样可以通过任务工厂TaskFactory类型来实现异步操作。

```
using System;
using System.Threading;
using System.Threading.Tasks;

namespace TaskFactory
{
    class Program
    {
        static void Main(string[] args)
        {
            ThreadPool.SetMaxThreads(1000, 1000);
            Task.Factory.StartNew(() => PrintMessage("Main Thread")
            Console.Read();
        }
        // 打印线程池信息
        private static void PrintMessage(String data)
        {
            int workthreadnumber;
            int iothreadnumber;

            // 获得线程池中可用的线程，把获得的可用工作者线程数量赋给workthr
            // 获得的可用I/O线程数量给iothreadnumber变量
            ThreadPool.GetAvailableThreads(out workthreadnumber, ou

            Console.WriteLine("{0}\n CurrentThreadId is {1}\n Curre
                data,
                Thread.CurrentThread.ManagedThreadId,
                Thread.CurrentThread.IsBackground.ToString(),
                workthreadnumber.ToString(),
                iothreadnumber.ToString());
        }
    }
}
```

运行结果：

file:///C:/Users/v-tozhi/documents/visual studio 2010/Projects/OfficeGetStarted/MultilThread/Task...

```
Main Thread
 CurrentThreadId is 10
 CurrentThread is background :True
 WorkerThreadNumber is:998
 IOThreadNumbers is: 1000
```

讲到这里CLR的工作者线程大致讲完了，希望也篇文章可以让大家对线程又有进一步的理解。在后面的一篇线程系列将谈谈CLR线程池的I/O线程。

[C# 线程处理系列]专题三：线程池中的I/O线程

上一篇文章主要介绍了如何利用线程池中的工作者线程来实现多线程，使多个线程可以并发地工作，从而高效率地使用系统资源。在这篇文章中将介绍如何用线程池中的I/O线程来执行I/O操作，希望对大家有所帮助。

目录：

一、I/O线程实现对文件的异步

二、I/O线程实现对请求的异步

三、总结

一、I/O线程实现对文件的异步

1.1 I/O线程介绍：

对于线程所执行的任务来说，可以把线程分为两种类型：工作者线程和I/O线程。

工作者线程用来完成一些计算的任务，在任务执行的过程中，需要CPU不间断地处理，所以，在工作者线程的执行过程中，CPU和线程的资源是充分利用的。

I/O线程主要用来完成输入和输出的工作的，在这种情况下， 计算机需要I/O设备完成输入和输出的任务，在处理过程中，CPU是不需要参与处理过程的，此时正在运行的线程将处于等待状态，只有等任务完成后才会有事可做， 这样就造成线程资源浪费的问题。为了解决这样的问题，可以通过线程池来解决这样的问题，让线程池来管理线程，前面已经介绍过线程池了， 在这里就不讲了。

对于I/O线程，我们可以将输入输出操作分成三个步骤：启动、实际输入输出、处理结果。用于实际输入输出可由硬件完成，并不需要CPU的参与，而启动和处理结果也可以不在同一个线程上，这样就可以充分利用线程资源。在.Net中通过以Begin开头的方法来完成启动，以End开头的方法来处理结果，这两个方法可以运行在不同的线程，这样我们就实现了异步编程了。

1.2 .Net中如何使用异步

注意：

其实当我们调用Begin开头的方法就是将一个I/O线程排入到线程池中（调用Begin开头的方法就把I/O线程加入到线程池中管理都是.Net机制帮我们实现的）。

（因为有些人会问什么地方用到了线程池了，工作者线程由线程池管理很好看出来，因为创建工作者线程直接调用**ThreadPool.QueueUserWorkItem**方法来把工作者线程排入到线程池中）。

在.net Framework中的FCL中有许多类型能够对异步操作提供支持，其中在FileStream类中就提供了对文件的异步操作的方法。

FileStream类要调用I/O线程要实现异步操作，首先要建立一个FileStream对象。

通过下面的构造函数来初始化FileStream对象实现异步操作(异步读取和异步写入):

public FileStream (string path, FileMode mode, FileAccess access, FileShare share,int bufferSize,bool useAsync)

其中path代表文件的相对路径或绝对路径，mode代表如何打开或创建文件，access代表访问文件的方式，share代表文件如何由进程共享，buffersize代表缓冲区的大小，useAsync代表使用异步I/O还是同步I/O,设置为true时，说明使用异步I/O.

下面通过代码来学习下异步写入文件：

```
using System;
using System.IO;
using System.Text;
using System.Threading;

namespace AsyncFile
{
    class Program
    {
        static void Main(string[] args)
        {
            const int maxsize = 100000;
            ThreadPool.SetMaxThreads(1000,1000);
            PrintMessage("Main Thread start");

            // 初始化FileStream对象
            FileStream filestream = new FileStream("test.txt", File

            //打印文件流打开的方式
            Console.WriteLine("filestream is {0} opened Asynchronou

            byte[] writebytes =new byte[maxsize];
            string writemessage = "An operation Use asynchronous me
            writebytes = Encoding.Unicode.GetBytes(writemessage);
            Console.WriteLine("message size is： {0} byte\n", write
            // 调用异步写入方法比信息写入到文件中
            filestream.BeginWrite(writebytes, 0, writebytes.Length,
            filestream.Flush();
            Console.Read();

        }

        // 当把数据写入文件完成后调用此方法来结束异步写操作
        private static void EndWriteCallback(IAsyncResult asyncResu
        {
            Thread.Sleep(500);
            PrintMessage("Asynchronous Method start");

            FileStream filestream = asyncResult.AsyncState as FileS
```

```
                // 结束异步写入数据
                filestream.EndWrite(asyncResult);
                filestream.Close();
            }

            // 打印线程池信息
            private static void PrintMessage(String data)
            {
                int workthreadnumber;
                int iothreadnumber;

                // 获得线程池中可用的线程，把获得的可用工作者线程数量赋给workthr
                // 获得的可用I/O线程数量给iothreadnumber变量
                ThreadPool.GetAvailableThreads(out workthreadnumber, ou

                Console.WriteLine("{0}\n CurrentThreadId is {1}\n Curre
                    data,
                    Thread.CurrentThread.ManagedThreadId,
                    Thread.CurrentThread.IsBackground.ToString(),
                    workthreadnumber.ToString(),
                    iothreadnumber.ToString());
            }
        }
    }
```

运行结果：

```
file:///c:/users/v-tozhi/documents/visual studio 2010/Projects/OfficeGetStarted/AsyncFile/bin/Deb...
Main Thread start
 CurrentThreadId is 9
 CurrentThread is background :False
 WorkerThreadNumber is:1000
 IOThreadNumbers is: 1000

filestream is  opened Asynchronously
message size is: 152 byte

Asynchronous Method start
 CurrentThreadId is 10
 CurrentThread is background :True
 WorkerThreadNumber is:1000
 IOThreadNumbers is: 999
```

从运行结果可以看出，此时是调用线程池中的I/O线程去执行回调函数的，同时在工程所的的bin\Debug文件目录下有生成一个text.txt文件，打开文件可以知道里面的内容正是你写入的。

下面演示如何从刚才的文件中异步读取我们写入的内容：

```
using System;
using System.IO;
```

```
using System.Text;
using System.Threading;

namespace AsyncFileRead
{
    class Program
    {
        const int maxsize = 1024;
        static byte[] readbytes = new byte[maxsize];
        static void Main(string[] args)
        {
            ThreadPool.SetMaxThreads(1000, 1000);
            PrintMessage("Main Thread start");

            // 初始化FileStream对象
            FileStream filestream = new FileStream("test.txt", File

            // 异步读取文件内容
            filestream.BeginRead(readbytes, 0, readbytes.Length, ne
            Console.Read();
        }

        private static void EndReadCallback(IAsyncResult asyncResul
        {
            Thread.Sleep(1000);
            PrintMessage("Asynchronous Method start");

            // 把AsyncResult.AsyncState转换为State对象
            FileStream readstream = (FileStream)asyncResult.AsyncSt
            int readlength = readstream.EndRead(asyncResult);
            if (readlength <=0)
            {
                Console.WriteLine("Read error");
                return;
            }

            string readmessage = Encoding.Unicode.GetString(readbyt
            Console.WriteLine("Read Message is :" + readmessage);
            readstream.Close();
        }

        // 打印线程池信息
        private static void PrintMessage(String data)
        {
            int workthreadnumber;
            int iothreadnumber;

            // 获得线程池中可用的线程，把获得的可用工作者线程数量赋给workthr
            // 获得的可用I/O线程数量给iothreadnumber变量
            ThreadPool.GetAvailableThreads(out workthreadnumber, ou

            Console.WriteLine("{0}\n CurrentThreadId is {1}\n Curre
                data,
```

```
                    Thread.CurrentThread.ManagedThreadId,
                    Thread.CurrentThread.IsBackground.ToString(),
                    workthreadnumber.ToString(),
                    iothreadnumber.ToString());
            }
        }
    }
```

运行结果：

file:///c:/users/v-tozhi/documents/visual studio 2010/Projects/OfficeGetStarted/AsyncFileRead/bin...

```
Main Thread start
 CurrentThreadId is 9
 CurrentThread is background :False
 WorkerThreadNumber is:1000
 IOThreadNumbers is: 1000

Asynchronous Method start
 CurrentThreadId is 10
 CurrentThread is background :True
 WorkerThreadNumber is:999
 IOThreadNumbers is: 1000

Read Message is :An operation Use asynchronous method to write message..........
.............
```

这里有个需要注意的问题：如果大家测试的时候， 应该把开始生成的text.txt文件放到该工程下bin\debug\目录下， 我刚开始的做的时候就忘记拷过去的， 读出来的数据长度一直为0（这里我犯的错误写下了，希望大家可以注意，也是警惕自己要小心。）

二、**I/O**线程实现对请求的异步

我们同样可以利用I/O线程来模拟对浏览器对服务器请求的异步操作，在.net类库中的WebRequest类提供了异步请求的支持，

下面就来演示下如何实现请求异步：

```
using System;
using System.Net;
using System.Threading;

namespace RequestSample
{
    class Program
    {
        static void Main(string[] args)
        {
            ThreadPool.SetMaxThreads(1000, 1000);
            PrintMessage("Main Thread start");

            // 发出一个异步Web请求
```

```
            WebRequest webrequest =WebRequest.Create("http://www.cr
            webrequest.BeginGetResponse(ProcessWebResponse, webrequ

            Console.Read();
        }

        // 回调方法
        private static void ProcessWebResponse(IAsyncResult result)
        {
            Thread.Sleep(500);
            PrintMessage("Asynchronous Method start");

            WebRequest webrequest = (WebRequest)result.AsyncState;
            using (WebResponse webresponse = webrequest.EndGetRespo
            {
                Console.WriteLine("Content Length is : "+webrespons
            }
        }

        // 打印线程池信息
        private static void PrintMessage(String data)
        {
            int workthreadnumber;
            int iothreadnumber;

            // 获得线程池中可用的线程，把获得的可用工作者线程数量赋给workthr
            // 获得的可用I/O线程数量给iothreadnumber变量
            ThreadPool.GetAvailableThreads(out workthreadnumber, ou

            Console.WriteLine("{0}\n CurrentThreadId is {1}\n Curre
                data,
                Thread.CurrentThread.ManagedThreadId,
                Thread.CurrentThread.IsBackground.ToString(),
                workthreadnumber.ToString(),
                iothreadnumber.ToString());
        }
    }
}
```

运行结果为：

file:///c:/users/v-tozhi/documents/visual studio 2010/Projects/OfficeGetStarted/AsyncFile/Request...

```
Main Thread start
 CurrentThreadId is 9
 CurrentThread is background :False
 WorkerThreadNumber is:1000
 IOThreadNumbers is: 1000

Asynchronous Method start
 CurrentThreadId is 13
 CurrentThread is background :True
 WorkerThreadNumber is:1000
 IOThreadNumbers is: 999

Content Length is : 68163
```

写到这里这篇关于I/O线程的文章也差不多写完了， 其实I/O线程还可以做很多事情，在网络(Socket)编程,web开发中都会用I/O线程，本来想写个Demo来展示多线程在实际的工作中都有那些应用的地方的， 但是后面觉得还是等多线程系列都讲完后再把知识一起串联起来做个Demo会好点，至于后面文章中将介绍下线程同步的问题。

[C# 线程处理系列]专题四：线程同步

目录：

一、线程同步概述

前面的文章都是讲创建多线程来实现让我们能够更好的响应应用程序，然而当我们创建了多个线程时，就存在多个线程同时访问一个共享的资源的情况，在这种情况下，就需要我们用到线程同步，线程同步可以防止数据（共享资源）的损坏。

然而我们在设计应用程序还是要尽量避免使用线程同步， 因为线程同步会产生一些问题：

1. 它的使用比较繁琐。因为我们要用额外的代码把多个线程同时访问的数据包围起来，并获取和释放一个线程同步锁，如果我们在一个代码块忘记获取锁，就有可能造成数据损坏。

2. 使用线程同步会影响性能，获取和释放一个锁肯定是需要时间的吧，因为我们在决定哪个线程先获取锁时候， CPU必须进行协调，进行这些额外的工作就会对性能造成影响

3. 因为线程同步一次只允许一个线程访问资源，这样就会阻塞线程，阻塞线程会造成更多的线程被创建，这样CPU就有可能要调度更多的线程，同样也对性能造成了影响。

所以在实际的设计中还是要尽量避免使用线程同步，因此我们要避免使用一些共享数据，例如静态字段。

二、线程同步的使用

2.1 对于使用锁性能的影响

上面已经说过使用锁将会对性能产生影响， 下面通过比较使用锁和不使用锁时消耗的时间来说明这点

```
using System;
using System.Diagnostics;
using System.Threading;

namespace InterlockedSample
{
    // 比较使用锁和不使用锁锁消耗的时间
    // 通过时间来说明使用锁性能的影响
    class Program
    {
        static void Main(string[] args)
        {
            int x = 0;
            // 迭代次数为500万
            const int iterationNumber = 5000000;
            // 不采用锁的情况
            // StartNew方法 对新的 Stopwatch 实例进行初始化，将运行时间属
            Stopwatch sw = Stopwatch.StartNew();
            for (int i = 0; i < iterationNumber; i++)
            {
                x++;
            }

            Console.WriteLine("Use the all time is :{0} ms", sw.Ela

            sw.Restart();
            // 使用锁的情况
            for (int i = 0; i < iterationNumber; i++)
            {
                Interlocked.Increment(ref x);
            }

            Console.WriteLine("Use the all time is :{0} ms", sw.Ela
            Console.Read();
        }
    }
}
```

运行结果（这是在我电脑上运行的结果）从结果中可以看出加了锁的运行速度慢了好多（慢了11倍 197/18 ）：

file:///C:/Users/v-tozhi/Documents/Visual Studio 2010/Projects/ThreadsSynchronous/Interlocked/...

```
Use the all time is :18 ms
Use the all time is :197 ms
```

2.2 Interlocked实现线程同步

Interlocked类提供了为多个线程共享的变量提供原子操作，当我们在多线程中对一个整数进行递增操作时，就需要实现线程同步。

因为增加变量操作（++运算符）不是一个原子操作，需要执行下列步骤：

1）将实例变量中的值加载到寄存器中。

2）增加或减少该值。

3）在实例变量中存储该值。

如果不使用 **Interlocked.Increment**方法，线程可能会在执行完前两个步骤后被抢先。然后由另一个线程执行所有三个步骤，此时第一个线程还没有把变量的值存储到实例变量中去，而另一个线程就可以把实例变量加载到寄存器里面读取了（此时加载的值并没有改变），所以会导致出现的结果不是我们预期的，相信这样的解释可以帮助大家更好的理解**Interlocked.Increment**方法和 原子性操作，

下面通过一段代码来演示下加锁和不加锁的区别（开始讲过加锁会对性能产生影响，这里将介绍加锁来解决线程同步的问题，得到我们预期的结果）：

不加锁的情况：

```
class Program
    {
        static void Main(string[] args)
        {     for (int i = 0; i < 10; i++)
            {
                Thread testthread = new Thread(Add);
                testthread.Start();
            }

            Console.Read();
        }

        // 共享资源
        public static int number = 1;

        public static void Add()
        {
            Thread.Sleep(1000);
            Console.WriteLine("the current value of number is:{0}",
        }
}
```

运行结果（不同电脑上可能运行的结果和我的不一样，但是都是得到不是预期的结果的）：

```
file:///C:/Users/v-tozhi/Documents/Visual Studio 2010/Projects/ThreadsSynchronous/Interlocked/...
the current value of number is:3
the current value of number is:2
the current value of number is:4
the current value of number is:6
the current value of number is:5
the current value of number is:7
the current value of number is:8
the current value of number is:11
the current value of number is:9
the current value of number is:10
```

为了解决这样的问题，我们可以通过使用 **Interlocked.Increment**方法来实现原子的自增操作。

代码很简单，只需要把++number改成Interlocked.Increment(ref number)就可以得到我们预期的结果了，在这里代码和运行结果就不贴了。

总之**Interlocked**类中的方法都是执行一次原子读取以及写入的操作的。

2.3 Monitor实现线程同步

对于上面那个情况也可以通过Monitor.Enter和Monitor.Exit方法来实现线程同步。C#中通过lock关键字来提供简化的语法(lock可以理解为Monitor.Enter和Monitor.Exit方法的语法糖),代码也很简单：

```
using System;
using System.Threading;

namespace MonitorSample
{
    class Program
    {
        static void Main(string[] args)
        {
            for (int i = 0; i < 10; i++)
            {
                Thread testthread = new Thread(Add);
                testthread.Start();
            }

            Console.Read();
        }

        // 共享资源
        public static int number = 1;

        public static void Add()
        {
            Thread.Sleep(1000);
            //获得排他锁
            Monitor.Enter(number);

            Console.WriteLine("the current value of number is:{0}",

            // 释放指定对象上的排他锁。
            Monitor.Exit(number);
        }
    }
}
```

运行结果当然是我们所期望的：

file:///C:/Users/v-tozhi/Documents/Visual Studio 2010/Projects/ThreadsSynchronous/Interlocked/...

```
the current value of number is:2
the current value of number is:3
the current value of number is:4
the current value of number is:5
the current value of number is:6
the current value of number is:7
the current value of number is:8
the current value of number is:9
the current value of number is:10
the current value of number is:11
```

在 Monitor类中还有其他几个方法在这里也介绍，只是让大家引起注意下，一个Wait方法，很明显Wait方法的作用是：释放某个对象上的锁以便允许其他线程锁定和访问这个对象。第二个就是TryEnter方法，这个方法与Enter方法主要的区别在于是否阻塞当前线程，当一个对象通过Enter方法获取锁，而没有执行Exit方法释放锁，当另一个线程想通过Enter获得锁时，此时该线程将会阻塞，直到另一个线程释放锁为止，而TryEnter不会阻塞线程。具体代码就不不写出来了。

2.4 ReaderWriterLock实现线程同步

如果我们需要对一个共享资源执行多次读取时，然而用前面所讲的类实现的同步锁都只允许一个线程允许，所有线程将阻塞，但是这种情况下肯本没必要堵塞其他线程， 应该让它们并发的执行，因为我们此时只是进行读取操作，此时通过ReaderWriterLock类可以很好的实现读取并行。

演示代码为：

```
using System;
using System.Collections.Generic;
using System.Threading;

namespace ReaderWriterLockSample
{
    class Program
    {
        public static List<int> lists = new List<int>();

        // 创建一个对象
        public static ReaderWriterLock readerwritelock = new Reade
        static void Main(string[] args)
        {
            //创建一个线程读取数据
            Thread t1 = new Thread(Write);
            t1.Start();
            // 创建10个线程读取数据
            for (int i = 0; i < 10; i++)
            {
                Thread t = new Thread(Read);
                t.Start();
            }
```

```
            Console.Read();

        }

        // 写入方法
        public static void Write()
        {
            // 获取写入锁，以10毫秒为超时。
            readerwritelock.AcquireWriterLock(10);
            Random ran = new Random();
            int count = ran.Next(1, 10);
            lists.Add(count);
            Console.WriteLine("Write the data is:" + count);
            // 释放写入锁
            readerwritelock.ReleaseWriterLock();
        }

        // 读取方法
        public static void Read()
        {
            // 获取读取锁
            readerwritelock.AcquireReaderLock(10);

            foreach (int li in lists)
            {
                // 输出读取的数据
                Console.WriteLine(li);
            }

            // 释放读取锁
            readerwritelock.ReleaseReaderLock();
        }
    }
}
```

运行结果：

file:///C:/Users/v-tozhi/Documents/Visual Studio 2010/Projects/ThreadsSynchronous/ReaderWriter...

```
Write the data is:6
6
6
6
6
6
6
6
6
6
6
```

三、总结

本文中主要介绍如何实现多线程同步的问题， 通过线程同步可以防止共享数据的损坏，但是由于获取锁的过程会有性能损失，所以在设计应用过程中尽量减少线程同步的使用。本来还要介绍互斥(Mutex), 信号量(Semaphore), 事件构造的， 由于篇幅的原因怕影响大家的阅读，所以这剩下的内容放在后面介绍的。

[C# 线程处理系列]专题五：线程同步——事件构造

引言：

其实这部分内容应该是属于专题四，因为这篇也是讲关于线程同步的，但是由于考虑到用户的阅读习惯问题，因为文章太长了，很多人不是很愿意看包括我也是这样的，同时也有和我说可以把代码弄成折叠的，这样就不会太长的，但是我觉得这样也不怎么便于阅读，因为我看别人的博客的时候，看到有代码是折叠起来的时候很多时候不愿意去点，并且点一下之后同样拉长文章的，然后就看到右边的滚动条变小了，本以为快看完了(意思快学到知识了)，一看滚动条后发现还有好长的内容很看， 所以就会给人一种不舒服的感觉吧(如果有和我一样的人的话，你肯定懂的是什么感觉的)。所以我把线程同步放到两篇文章里面来说，其实放到两篇文章里面也有一定原因的， 前面讲的线程同步主要是用户模式的(CLR Via C# 一书中是这么定义的，书中说到线程同步分两种：一、用户模式构造 二、内核模式构造，第一次看的时候不是很理解两个名词是什么意思的，我一般理解东西是采用把东西拆分来理解，理解拆分的各个部分后再合起来理解内容的，现在我对着两个的理解是——用户模式构造：对于内核模式构造（指的的是构造操作系内核对象），我们使用类（.net Framework中的类，如 AutoResetEvent, Semaphore类）的方法来实现线程同步，其实内部是调用操作系统的内核对象来实现的线程同步，此时就会导致线程从托管代码到为内核代码，然而用户模式构造，没有调用操作系统内核对象，线程只是在用户的托管代码上执行的)，对于用户模式构造和内核模式的构造只是我自己的理解的， 如果有更好的理解方式可以留言告诉下我， 这样我们可以一起讨论和学习了。

目录：

一、WaitHandle基类介绍

二、事件(Event)类实现线程同步

三、信号量(Semapyore)类实现线程同步

四、互斥体(Mutex)实现线程同步

一、WaitHandle基类介绍

System.Threading命名空间中提供了一个**WaitHandle** 的抽象基类，此类就是包装了一个Windows内核对象的句柄（句柄可以理解为标示了对象实例的一个数字，具体大家可以查看资料深入理解下的，在这里只是提出理解句柄也是很重要的），在.net Framework中提供了从WaitHandle类中派生的类（我正是用这些派生类在我们的代码中实现线程同步的）。它们的一个继承关系为

WaitHandle

EventWaitHandle

AutoResetEvent

ManualResetEvent

Semaphore

Mutex

当我们在使用 **AutoResetEvent**,**ManualResetEvent,****Semaphore,****Mutex**这些类的时候，用构造函数来实例化这些类的对象时，其内部都调用了Win32 CreateEvent或CreateEvent函数，或CreateSemaphore或者CreateMutex函数，这些函数调用返回的句柄值都保存在WaitHandle基类定义的SafeWaitHandle字段中。

二、事件**(Event)**类实现线程同步

2.1 AutoResetEvent （自动重置事件）

先讲讲AutoresetEvent类的构造函数，其定义为：

public AutoResetEvent(bool initialState);

构造函数中用一个bool 类型的初始状态来设置AutoResetEvent对象的状态，如果要将AutoResetEvent对象的初始状态设置为终止，则传入bool值为true，若要设置非终止，就传入false。

WaitOne方法定义：

public virtual bool WaitOne(int millisecondsTimeout）;该方法用来阻塞线程，当在指定的时间间隔还没有收到一个信号时，将返回false。

调用Set方法发信号来释放等待线程。在使用过程中WaitOne方法和Set方法都是成对出现的， 一个用于阻塞线程，等待信号，一个用来释放等待线程(就是说调用set方法来发送一个信号，此时WaitOne接受到信号，就释放阻塞的线程，线程就可以继续运行)

线程通过调用AutoResetEvent的WaitOne方法来等待信号，如果AutoResetEvent对象为非终止状态，则线程被阻止，等到线程调用Set方法来恢复线程执行。如果AutoResetEvent为终止状态时，则线程不会被阻止，此时AutoResetEvent将立即释放线程并返回为非终止状态（指出有线程在使用资源的一种状态）。

下面通过通过一个例子来演示下AutoResetEvent的使用：

```
using System;
using System.Threading;

namespace KenelMode
{
    class Program
    {
        // 初始化自动重置事件，并把状态设置为非终止状态
        // 如果这里把初始状态设置为True时，
        // 当调用WaitOne方法时就不会阻塞线程,看到的输出结果的时间就是一样的了
        // 因为设置为True时，表示此时已经为终止状态了。
        public static AutoResetEvent autoEvent = new AutoResetEvent
        static void Main(string[] args)
        {
            Console.WriteLine("Main Thread Start run at: " +DateTim
            Thread t = new Thread(TestMethod);
            t.Start();

            // 阻塞主线程3秒后
            // 调用 Set方法释放线程，使线程t可以运行
            Thread.Sleep(3000);

            // Set 方法就是把事件状态设置为终止状态。
            autoEvent.Set();
            Console.Read();
        }

        public static void TestMethod()
        {
            autoEvent.WaitOne();

            // 3秒后线程可以运行，所以此时显示的时间应该和主线程显示的时间相
            Console.WriteLine("Method Restart run at: " + DateTime.
        }
    }
}
```

运行结果（从运行结果看确实是过了一秒后在TestMethod方法中的语句）：

```
file:///C:/Users/v-tozhi/Documents/Visual Studio 2010/Projects/KenelMode/KenelMode/bin/Debu...
Main Thread Start run at: 1:54:41 PM
Method Restart run at: 1:54:44 PM
```

上面中用到的是没有带参数的WaitOne方法，该方法表示无限制阻塞线程，直到收到一个事件为止（通过Set方法来发送一个信号），同时我们也可以设置堵塞线程的事件，当超时时，线程将不阻塞直接运行（尽管此时没有通过Set来发送一个信号，线程照样运行，只是WaitOne方法返回的的值不一样）。

bool WaitOne(int millisecondsTimeout) 收到信号时返回为True,没收到信号返回为false。

看完下面的代码你可能会形象理解WaitOne(millisecondsTimeout)方法的使用的：

```
using System;
using System.Threading;

namespace KenelMode
{
    class Program
    {
        // 初始化自动重置事件，并把状态设置为非终止状态
        // 如果这里把初始状态设置为True时，
        // 当调用WaitOne方法时就不会阻塞线程,看到的输出结果的时间就是一样的了
        // 因为设置为True时，表示此时已经为终止状态了。
        public static AutoResetEvent autoEvent = new AutoResetEvent
        static void Main(string[] args)
        {
            Console.WriteLine("Main Thread Start run at: " +DateTin
            Thread t = new Thread(TestMethod);
            t.Start();

            // 阻塞主线程1秒后
            // 调用 Set方法释放线程，使线程t可以运行
            Thread.Sleep(3000);

            // Set 方法就是把事件状态设置为终止状态。
            autoEvent.Set();
            Console.Read();
        }

        public static void TestMethod()
        {
            if (autoEvent.WaitOne(2000))
            {
                Console.WriteLine("Get Singal to Work");
                // 3秒后线程可以运行，所以此时显示的时间应该和主线程显示的时
                Console.WriteLine("Method Restart run at: " + DateT
            }
            else
            {
                Console.WriteLine("Time Out to work");
                Console.WriteLine("Method Restart run at: " + DateT
            }
        }
    }
}
```

运行结果：

file:///C:/Users/v-tozhi/Documents/Visual Studio 2010/Projects/KenelMode/KenelMode/bin/Debu...

```
Main Thread Start run at: 2:15:19 PM
Time Out to work
Method Restart run at: 2:15:21 PM
```

同时这里可以把Thread.Sleep(3000)改成Thread.Sleep(1000)的时候，就是说AutoResetEvent对象在超时之前就接到信号了， 此时WaitOne(2000)放回的值就是True,就得到的是Get Singal to Work, 之间的事件间隔当然也是1秒了，在这里结果就不贴了。

2.2 ManualResetEvent(手动重置事件)

WaitOne while the AutoResetEvent is in the signaled state, the thread does not block." data-guid="89f5b5007b8c3bb9a7bd377a380ad62c">ManualResetEvent的使用和AutoResetEvent的使用很类似，因为他们都是从EventWaitHandle类派生的，不过他们还是有点区别:

WaitOne while the AutoResetEvent is in the signaled state, the thread does not block." data-guid="89f5b5007b8c3bb9a7bd377a380ad62c">AutoResetEvent 为终止状态时线程调用 WaitOne，则线程不会被阻止。AutoResetEvent releases the thread immediately and returns to the non-signaled state." data-guid="6b52aed4c2b560eeba356721b90fc103">**AutoResetEvent** 将立即释放线程并返回到非终止状态**,**当再次调用**WaitOne**状态时线程会被阻止

AutoResetEvent releases the thread immediately and returns to the non-signaled state." data-guid="6b52aed4c2b560eeba356721b90fc103">这里请注意如果**AutoResetEvent**初始为非终止状态时， 调用**WaitOne(int millisecondsTimeout)**方法后并不会把状态返回为终止状态，此时还是非终止的，调用**WaitOne**方法自动改变状态只针对初始状态为终止状态时有效。

AutoResetEvent releases the thread immediately and returns to the non-signaled state." data-guid="6b52aed4c2b560eeba356721b90fc103">然而ManualResetEvent初始状态为终止状态时时调用WaitOne，则线程同样不会被阻止，但是**ManualResetEvent**的状态不会发生改变（当我再次调用**WaitOne**方法是一样不会阻止线程），需要我们手动终止**()**

下面通过一段代码来说明两者的区别：

```
using System;
using System.Threading;

namespace ManualResetEventSample
{
    class Program
    {
        // 初始化自动重置事件，并把状态设置为终止状态
        public static AutoResetEvent autoEvent = new AutoResetEvent

        ////public static ManualResetEvent autoEvent = new ManualRe
        static void Main(string[] args)
        {
            Console.WriteLine("Main Thread Start run at: " + DateTi
            Thread t = new Thread(TestMethod);
            t.Start();
            Console.Read();
        }

        public static void TestMethod()
        {
            // 初始状态为终止状态，则第一次调用WaitOne方法不会堵塞线程
            // 此时运行的时间间隔应该为0秒，但是因为是AutoResetEvent对象
            // 调用WaitOne方法后立即把状态返回为非终止状态。
            autoEvent.WaitOne();
            Console.WriteLine("Method start at : "+ DateTime.Now.To

            // 因为此时AutoRestEvent为非终止状态，所以调用WaitOne方法后将
            // 所以下面语句的和主线程中语句的时间间隔为1秒
            // 当时 ManualResetEvent对象时，因为不会自动重置状态
            // 所以调用完第一次WaitOne方法后状态仍然为非终止状态,所以再次调
            // 如果没有设置超时时间的话，下面这行语句将不会执行
            autoEvent.WaitOne(1000);
            Console.WriteLine("Method start at : " + DateTime.Now.T

        }
    }
}
```

运行结果：

```
file:///C:/Users/v-tozhi/Documents/Visual Studio 2010/Projects/KenelMode/ManualResetEventSa...
Main Thread Start run at: 3:08:11 PM
Method start at : 3:08:11 PM
Method start at : 3:08:12 PM
```

如果你把创建事件为手动重置事件ManualResetEvent时，得到的运行结果就会下面这样：

```
file:///C:/Users/v-tozhi/Documents/Visual Studio 2010/Projects/KenelMode/Mar
Main Thread Start run at: 3:10:42 PM
Method start at : 3:10:42 PM
Method start at : 3:10:42 PM
```

2.3 跨进程之间同步

内核模式的构造可同步在同一台机器上的不同进程中运行的线程，所以我们同样可以使用 AutoResetEvent实现不同进程中运行的线程同步，但是此时需要对AutoResetEvent进行命名，但是AutoResetEvent只提供带一个参数的构造函数的，此时应该如何去实现不同进程中的线程同步的呢？

其实是有解决办法的，因为AutoResetEvent是继承自EventWaitHandle类的，EventWaitHandle类有多个构造函数的

除了之前的方法创建AutoResetEvent对象外，

还可以通过EventWaitHandle AutoEvent = new EventWaitHandle (false, EventResetMode.Auto);这样的方法来构造AutoResetEvent对象，通过

EventWaitHandle autoEvent = new EventWaitHandle (false, EventResetMode.Auto,"My");方式就可以指定名称了

下面一段代码演示如何实现跨不同进程中的线程同步：

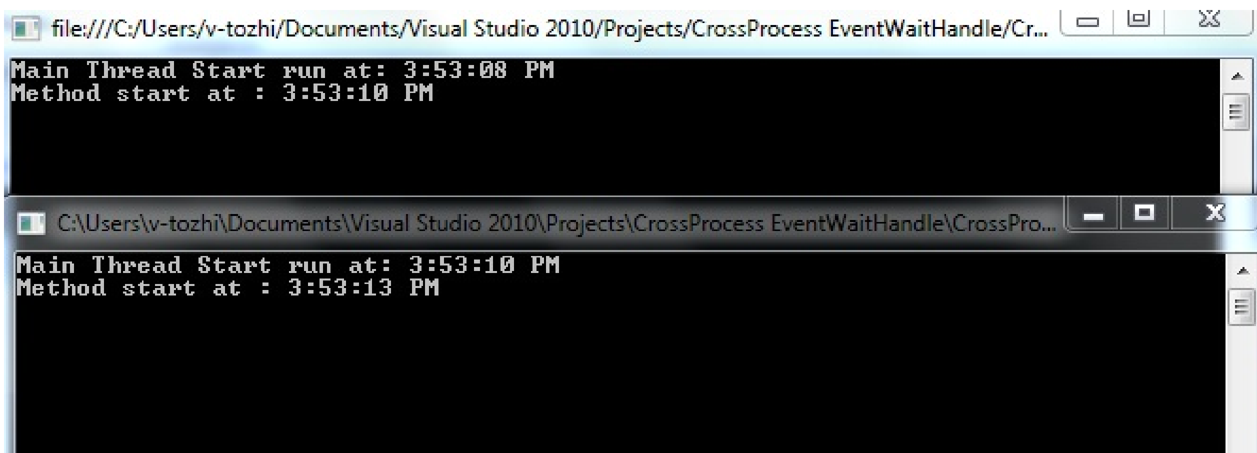
```
using System;
using System.Threading;

namespace CrossProcess_EventWaitHandle
{
    class Program
    {
        public static EventWaitHandle autoEvent = new EventWaitHanc
        static void Main(string[] args)
        {
            Console.WriteLine("Main Thread Start run at: " + DateTi
            Thread t = new Thread(TestMethod);

            // 为了有时间启动另外一个线程
            Thread.Sleep(2000);
            t.Start();
            Console.Read();
        }

        public static void TestMethod()
        {
            // 进程一：显示的时间间隔为2秒
            // 进程二中显示的时间间隔为3秒
            // 因为进程二中AutoResetEvent的初始状态为非终止的
            // 因为在进程一中通过WaitOne方法的调用已经把AutoResetEvent的
            autoEvent.WaitOne(1000);
            Console.WriteLine("Method start at : "+ DateTime.Now.To
        }
    }
}
```

运行结果：



本来打算在一篇文章里面讲述内核模式构造的，写着写着滚动条又变很小了，为了大家的阅读，我把信号量和互斥体放在后面一篇文章里面讲吧，相信后面的内容会很好理解的，因为后面两个类的使用和这篇中讲到的使用很类似，好歹都是继承WaitHandle类的。

[C# 线程处理系列]专题六：线程同步——信号量和互斥体

也不多说了，直接进入主题了

一、信号量（**Semaphore**）

信号量（Semaphore）是由内核对象维护的int变量，当信号量为0时，在信号量上等待的线程会堵塞，信号量大于0时，就解除堵塞。当在一个信号量上等待的线程解除堵塞时，内核自动会将信号量的计数减1。在.net 下通过Semaphore类来实现信号量同步。

Semaphore类限制可同时访问某一资源或资源池的线程数。WaitOne method, which is inherited from the WaitHandle class, and release the semaphore by calling the Release method.">线程通过调用 WaitOne方法将信号量减1，并通过调用 Release方法把信号量加1。

先说下构造函数：

public Semaphore(int initialCount,int maximumCount);通过两个参数来设置信号的初始计数和最大计数。

下面通过一段代码来演示信号量同步的使用：

```
class Program
    {
        // 初始信号量计数为0，最大计数为10
        public static Semaphore semaphore =new Semaphore(0,10);
        public static int time = 0;
        static void Main(string[] args)
        {
            for (int i = 0; i < 5; i++)
            {
                Thread test = new Thread(new ParameterizedThreadSta

                // 开始线程，并传递参数
                test.Start(i);
            }

            // 等待1秒让所有线程开始并阻塞在信号量上
            Thread.Sleep(500);

            // 信号量计数加4
            // 最后可以看到输出结果次数为4次
            semaphore.Release(4);
            Console.Read();
        }

        public static void TestMethod(object number)
        {
            // 设置一个时间间隔让输出有顺序
            int span = Interlocked.Add(ref time, 100);
            Thread.Sleep(1000 + span);

            //信号量计数减1
            semaphore.WaitOne();

            Console.WriteLine("Thread {0} run ", number);
        }
    }
```

运行结果：

```
file:///C:/Users/v-tozhi/Documents/Visual Studio 2010/Projects/KenelMode/SemaphoreSample/bi...
Thread 0 run
Thread 1 run
Thread 2 run
Thread 3 run
```

同样信号量也可以实现进程中线程的同步，同样也是通过对信号量命名来实现的，

通过调用public Semaphore(int initialCount,int maximumCount,string name);该构造函数多传入一个信号量名来实现

下面一段实例代码来演示下：

```
using System;
using System.Threading;

namespace SemaphoreSample
{
    class Program
    {
        // 初始信号量计数为4，最大计数为10
        public static Semaphore semaphore =new Semaphore(4,10,"My")
        public static int time = 0;
        static void Main(string[] args)
        {
            for (int i = 0; i < 3; i++)
            {
                Thread test = new Thread(new ParameterizedThreadSta

                // 开始线程，并传递参数
                test.Start(i);
            }

            // 等待1秒让所有线程开始并阻塞在信号量上
            Thread.Sleep(1000);

            Console.Read();
        }

        public static void TestMethod(object number)
        {
            // 设置一个时间间隔让输出有顺序
            int span = Interlocked.Add(ref time, 500);
            Thread.Sleep(1000 + span);

            //信号量计数减1
            semaphore.WaitOne();

            Console.WriteLine("Thread {0} run ", number);
        }
    }
}
```

运行结果：

```
file:///C:/Users/v-tozhi/Documents/Visual Studio 2010/Projects/KenelMode/SemaphoreSample/bi...
Thread 0 run
Thread 1 run
Thread 2 run
```

```
C:\Users\v-tozhi\Documents\Visual Studio 2010\Projects\KenelMode\SemaphoreSample\bin\Deb...
Thread 0 run
```

从运行结果中可以看出， 第二个进程值运行了一行语句， 因为我们设置的初始信号计数为4，每运行一个线程，信号计数通过调用WaitOne方法减1,所以第二个进行一开始信号计数为1而不是进程一中的4，如果我们把信号计数后面的name参数去除的话，此时第二个进程和第一个进程中的结果应该是一样的（因为此时没有进行不同进程中线程的同步）。

二、互斥体（**Mutex**）

同样互斥体也是同样可以实现线程之间的同步和不同进程中线程的同步的

先看看线程之间的同步的例子吧（在这里我也不多做解释了，因为他们之间的使用很类似，直接贴出代码）：

```
class Program
    {
        public static Mutex mutex = new Mutex();
        public static int count;

        static void Main(string[] args)
        {
            for (int i = 0; i < 10; i++)
            {
                Thread test = new Thread(TestMethod);

                // 开始线程，并传递参数
                test.Start();
            }

            Console.Read();
        }

        public static void TestMethod()
        {
            mutex.WaitOne();
            Thread.Sleep(500);
            count++;
            Console.WriteLine("Current Cout Number is {0}", count);
            mutex.ReleaseMutex();
        }
    }
```

运行结果：

```
file:///C:/Users/v-tozhi/Documents/Visual Studio 2010/Projects/KenelMode/MutexSample/bin/De...
Current Cout Number is 1
Current Cout Number is 2
Current Cout Number is 3
Current Cout Number is 4
Current Cout Number is 5
Current Cout Number is 6
Current Cout Number is 7
Current Cout Number is 8
Current Cout Number is 9
Current Cout Number is 10
```

实现进程间同步：

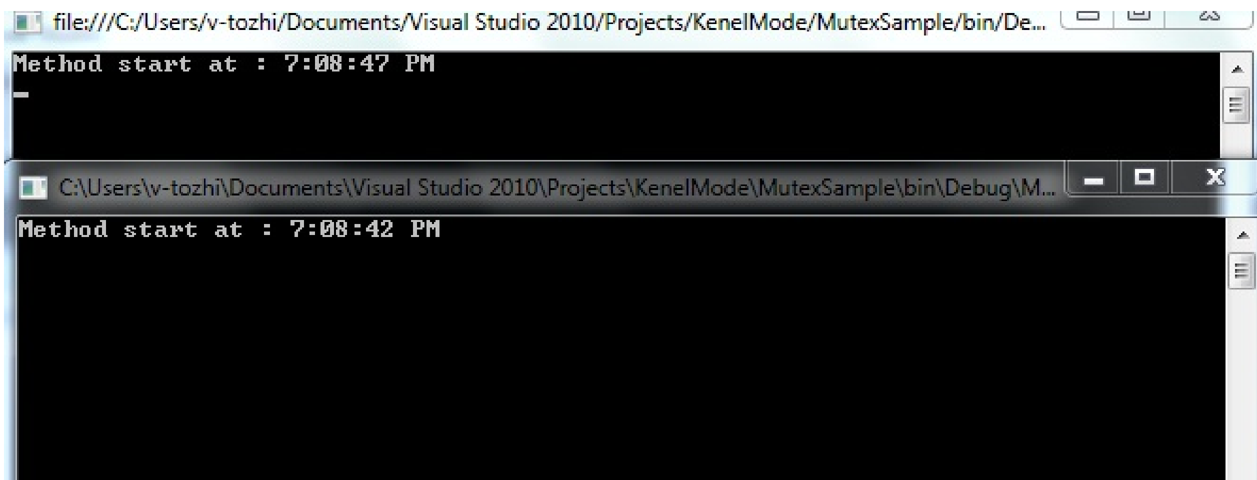
```
class Program
    {
        public static Mutex mutex = new Mutex(false,"My");

        static void Main(string[] args)
        {
            Thread t = new Thread(TestMethod);
            t.Start();

            Console.Read();
        }

        public static void TestMethod()
        {
            mutex.WaitOne();
            Thread.Sleep(5000);
            Console.WriteLine("Method start at : " + DateTime.Now.T
            mutex.ReleaseMutex();
        }
    }
```

运行结果：



从运行结果看出两个进程之间的时间间隔为5秒，当我们把构造函数中命名参数去掉时就可以看出差别了。

到这里多线程处理基本上讲完，这个系列也只是一个入门，真真要好好掌握多线程，还是要在项目中多去实战的。接下来我可能会做一个小的例子的，大概的思路是实现一个文件的下载的这样的例子。如果大家有什么好的例子来运用多线程的知识的话，可以留言给我，我也会尽量去实现（如果不会的话，这样也可以促使我去学习），实现后也会和大家分享的。

[C# 多线程处理系列专题七——对多线程的补充

因为有些人可能会疑惑，将了这么多多线程，到底在实际的应用上有什么作用的呢？ 这里我在这里用多线程简单实现了一个文件的下载的功能。

服务器端页面：

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Default.a

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "htt

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title></title>
</head>
<body>
    <form id="form1" runat="server">
    <div>

    <asp:Image ID="Image1" runat="server" ImageUrl="~/Images/1.gif'

        说明： CLR Via C#
        </div>

    </form>
</body>
</html>
```

服务器页面只是一个简单显示需要下载文件的一些信息，这里通过Handler.ashx来处理文件的下载，把文件的转化为二进制字节写入到输出流中，具体实现代码为：

```
 public class Handle : IHttpHandler
    {
        public void ProcessRequest(HttpContext context)
        {
            HttpResponse response = context.Response;
            HttpRequest request = context.Request;
            FileStream fileStream = null;
            byte[] buffer = new Byte[10240];
            int length;

            // 剩余的字节大小
            // 因为这里采取的是每次写入10240字节到输出流中
            long readToData;
            try
            {
                string filename = "CLR via CSharp 3rd edition.pdf",
```

```
                string filepath = HttpContext.Current.Server.MapPat

                fileStream = new FileStream(filepath, FileMode.Open
                readToData = fileStream.Length;
                while (readToData > 0)
                {
                    // 实际读取的字节大小
                    length = fileStream.Read(buffer, 0, buffer.Leng
                    // 把读取到的字节写入输出流中
                    response.OutputStream.Write(buffer, 0, length);
                    response.Flush();
                    readToData = readToData - length;
                }
            }
            catch (Exception ex)
            {
                response.Write("Error:" + ex.Message);
            }
            finally
            {
                if (fileStream != null)
                {
                    fileStream.Close();
                }

                response.End();
            }
        }

        public bool IsReusable
        {
            get
            {
                return false;
            }
        }
    }
```

这里牵涉到HttpHandle对象问题，这个对象在Asp.net中是真正处理数据的对象，后面如果有时间也和大家分享下深入理解Asp.net系列，主要是介绍在Asp.net中一些核心对象为我们默默做的一些事情，在这里也不详细介绍HttpHandle对象了， 这个示例中主要通过这个类来对文件的处理，把文件的二进制字节写入到输出流中， 客户端在从输出流中读取字节，然后保存为文件（其实文件也就是“流”）。

客户端：

客户端建立了一个WinForm窗口，通过WebBrower控件（就是在WinForm程序中显示网页的控件）来连接服务器页面，当按下下载按钮后，通过线程池线程来执行下载方法。主要代码为：

```
public void DownLoad(object state)
        {
            // 计时对象
            Stopwatch sw = Stopwatch.StartNew();

            HttpWebRequest request;
            HttpWebResponse response;
            Stream stream;

            // 下载下来的保存的地址
            string savepath = "D:\\Download.pdf";
            FileStream savestream = new FileStream(savepath, FileMo
            try
            {
                // 发出请求
                request = (HttpWebRequest)HttpWebRequest.Create(url

                // 获得回应对象
                response = (HttpWebResponse)request.GetResponse();

                // 获得回应流
                stream = response.GetResponseStream();
                byte[] bytes = new byte[10240];
                int readsize;

                // 每次都读取10240字节
                // 采用的是同步读取方法
                // 计算耗费的时间
                readsize = stream.Read(bytes, 0, bytes.Length);
                while (readsize > 0)
                {
                    savestream.Write(bytes, 0, readsize);
                    readsize = stream.Read(bytes, 0, bytes.Length);
                }

                sw.Stop();
                MessageBox.Show("下载耗时为：" + sw.Elapsed.ToString(
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message, "Error");
            }
            finally
            {
                savestream.Close();
            }
        }
```

这样就利用线程池线程简单完成了客户端下载服务器端文件的功能，并且使用线程池线程这样不会主线程，从而导致在下载文件时，界面同样可以操作，如果不采用多线程操作的话将会在下载过程导致界面“卡死”现象，这样就会给用户带来不好的用户体验。

其实本来还想做复杂点的， 开始想实现的功能，是服务器端断点续传，然后客户端多线程下载的功能的，这个示例中只用到了一个线程池线程来完成下载任务，本来想通过执行多个线程池线程来完成下载任务的， 每个线程只负责一部分的读取工作， 然后把每个线程中读取的字节合并起来就是完整的文件字节了，但是这里遇到一个问题，怎么在服务器端实现续传的功能的， 客户端通过AddRange方法来发出部分读取请求，然后服务器端就要对请求头Range进行解析的，实现原理我还是清楚，但是在做的过程中还是出现了问题。所以这里只能分享一个简单的下载文件的功能给大家了， 至于多线程的下载和断点续传和大文件的上传等问题，等我学习了再和大家分享， 如果有大牛可以帮助我解决服务端断点续传的问题的话，欢迎留言。

源文件下载链接：http://files.cnblogs.com/zhili/FileServer.zip （下载下来后只需要在服务器端Resources文件夹下添加一个文件就可以运行了）

C#网络编程系列

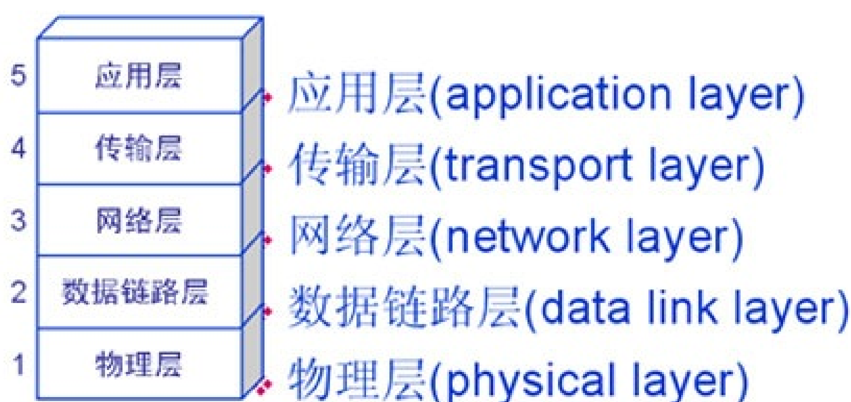
[C#网络编程系列]专题一：网络协议简介

因为这段时间都在研究C#网络编程的一些知识， 所以在这里把我学习到的在这里和大家分享下的，这样既可以达到分享的目的也可以让大家监督我，如果有什么地方理解错了，还请大家不吝赐教的。

很多人写网络编程这快都没有怎么讲网络中的协议，然而我觉得既然是网络编程肯定要介绍下网络编程中一些协议的，这样可以让更好的梳理网络编程的知识的，所以我在这系列中会用两个专题去讲协议，第一个专题简单介绍网络分层以及各层之间如何通信的只是，第二专题将会介绍下应用层协议——Http协议，了解这个不仅对网络编程有个理论基础，也可以帮助更好地理解Web(Asp.net)的开发。

一、网络分层

网络上的计算机之所以可以互相通信，是因为它们之间都遵守互相都可以"认识"的互联网协议（就如同人交流一样，两个人能够交流，互相必须知道对象的语言），互联网上的计算机互相通信又归根于网络中层与层之间的通信，OSI模型把网络通信分成七层：物理层、数据链路层、网络层、传输层、会话层、表示层和应用层，对于开发网络应用人员来说，一般把网络分成五层，这样比较容易理解。这五层为：物理层、数据链路层、网络层、传输层和应用层（最顶层），下面是一张网络分层的图片（来源于网络）：



二、各层的协议

网络中的计算机互相通信就是实现了层与层之间的通信，要实现层与层之间的通信，则各层都要遵守规则，这样才能完成更好的通信， 我们就把它们之间遵守的规则就叫个"协议"，然而网络上的五层之间遵守的协议不一样，每层都有各自的协议。下面就由下至上的讲述每层的协议

2.1 物理层协议

物理层是五层模型中的最底层，物理层为计算机之间的数据通信提供了传输媒体和互连设备，为数据传输提供了可靠的环境，媒体包括电缆、光纤、无线信道等，互连设备指是计算机和调制解调器之间的互连设备，如各种插头、插座等。该层的作用是透明的传输比特流（即二进制流），为数据链路层提供一个传输原始比特流的物理连接

2.2 数据链路层

数据链路层是模型中的第2层，该层对接受到物理层传输过来的比特流进行分组，一组电信号构成的数据包，就叫做"帧"，数据链链路层就是来传输以"帧"为单位的数据包，把数据传递给上一层（网络层），帧数据由两部分组成：帧头和帧数据，帧头包括接受方物理地址（就是网卡的地址）和其他的网络信息，帧数据就是要传输的数据体。数据帧的最长为1500字节，如果数据很长，就必须分割成多个帧进行发送。

2.3 网络层

该层通过寻址（寻址地址）来建立两个节点之间的连接，大家都知道我们的电脑连接上网络后都一个IP地址，我们可以通过IP地址来确定不同的计算机是否在同一个子网路。如果我们的电脑连接上网络后就有两种地址：物理地址和网络地址（IP地址），网络上的计算机要通信，必须要知道通信的计算机“在哪里”， 首先通过网络地址来判断是否处于同一个子网络，然后再对物理地址（MAC）地址进行处理，从而准确确定要通信计算机的位置。

在网络层中有我们熟悉的IP协议（即规定网络地址的协议），目前广泛采用的是IP协议第四版（IPv4）,这个版本规定，网络地址由32位二进制位组成。我们可以自己配置IP地址也可以自动获得的方式得到IP地址，Ip地址分成两部分，前24位代表网络，后8位代表主机号， 如192.168.254.1和192.168.254.2就处于同一个子网络里，因为这两个IP地址的前24位相同。

网络层中以IP数据包的形式来传递数据，IP数据包也包括两部分：头（Head）和数据(Data)，IP数据包放进数据帧中的数据部分进行传输。

2.4 传输层

通过MAC和IP地址，我们可以找到互联网上任意两台主机来建立通信。然而这里有一个问题，找到主机后，主机上有很多程序都需要用到网络，比如说你在一边听歌和好用QQ聊天， 当网络上发送来一个数据包时， 是怎么知道它是表示聊天的内容还是歌曲的内容的， 这时候就需要一个参数来表示这个数据包是发送给那个程序（进程）来使用的，这个参数我们就叫做端口号，主机上用端口号来标识不同的程序（进程），端口是0到65535之间的一个整数，0到1023的端口被系统占用，用户只能选择大于1023的端口。

传输层的功能就是建立端口到端口的通信，网络层就是建立主机与主机的通信，这样如果我们确定了主机和端口，这样就可以实现程序之间的通信了。我们所说的Socket编程就是通过代码来实现传输层之间的通信。因为初始化Socket类对象要指定IP地址和端口号。

在传输层有两个非常重要的协议：UDP 协议和TCP协议

采用UDP协议话传输的就是UDP数据包，同样UDP数据包也由头和数据两部分组成，头部分主要标识了发送端口和接受端口，数据部分就是具体的内容信息。同样**UDP**数据包是放入**IP**数据包中的**"**数据**"**部分，**IP**数据包再放入数据帧中在网络上传输。

由于UDP协议的可靠性差（数据发送后无法确定对方是否收到），所以又定义了一个可靠性高的协议——TCP协议，TCP协议采取了握手的方式要确保对方收到了数据。

2.5 应用层

应用层是模型中的最顶层，是用户与网络的接口，该层通过应用程序来完成网络用户的应用需求。该层的数据放在TCP数据包的数据部分，该层定义了一个很重要的协议——Http协议，我们一般的Web开发都是基于应用层的开发， 所以后面专题将会和大家介绍下Http协议。理解Http协议可以帮助我们理解Asp.net的请求响应模型以及帮助我们自定义发出请求和自定义服务器。

三、总结

现在通过一个简单的访问网页的例子来说明网络中的通信。

当我们在浏览器中输入www.baidu.com时，这意味着浏览器要向百度发送一个网页数据包，要发送数据包，需要知道对方的IP地址，这里我们只知道网址为www.baidu.com,却不知道IP地址，此时应用层协议DNS协议会帮我们把网址解析为IP地址，此时会发送一个DNS数据包给DNS服务器，DNS服务器再做出响应来告诉我们百度的IP地址为220.181.111.147，这样我们就知道百度（我们需要通信的主机）的IP地址。

应用层：

浏览网页采用的是HTTP 协议，HTTP数据包会嵌入在TCP数据包中，此时我们发送的HTTP数据包内容为：

```
GET http://www.baidu.com/ HTTP/1.1
Accept: application/x-ms-application, image/jpeg, application/xaml-
Accept-Language: en-US
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.1; WOW6
Accept-Encoding: gzip, deflate, peerdist
Proxy-Connection: Keep-Alive
Host: www.baidu.com
Cookie: BDSFRCVID=H1K_JgC2l434o0a3SlYrhIyDwFLxPM7C3J; H_BDCLCKID_SF
X-P2P-PeerDist: Version=1.0
```

传输层：TCP数据包需要设置端口，接收方（百度）的Http端口默认是80，本机的端口是一个1024-65535之间的随机整数，这里假设为1025，这样TCP数据包由标头（标识着发方和接收方的端口信息）+HTTP数据包，这样TCP数据包再嵌入IP数据包中在网络上传送

网络层：

IP数据包需要知道双方的IP地址，本机IP地址假定为192.168.1.5，接受方IP地址为220.181.111.147（百度），这样IP数据包由头部（IP地址信息）+TCP数据包，

数据链路层：

IP数据包嵌入到数据帧（以太网数据包）中，以太网数据包需要知道双方的MAC（物理地址），发送方为本机的网卡地址，接受方为网关192.168.1.1的MAC地址（通过ARP地址解析协议得到的）。这样数据帧由头部（MAC地址）+IP数据包组成。

经过多个网关的转发到百度服务器220.181.111.147，服务器接受到发送过来的以太网数据包，然后再从以太网数据包中提取IP数据包——>TCP数据包——>HTTP数据包，最后服务器做出"HTTP响应"，再用TCP协议发回给客户端（浏览器），浏览器同样的过程读取到HTTP响应的内容（HTTP响应数据包），然后浏览器对接受到的HTML页面进行解析，把网页显示出来呈现给用户，这样就完成了一次网络通信了。

后面一个专题将对HTTP协议进行详细的介绍。

[C#]网络编程系列专题二：HTTP协议详解

我们在用Asp.net技术开发Web应用程序后，当用户在浏览器输入一个网址时就是再向服务器发送一个HTTP请求，此时就使用了应用层的HTTP协议，在上一个专题我们简单介绍了网络协议的知识，主要是为了后面讲HTTP协议做一个铺垫的，只有对HTTP协议有一个清楚的认识，这样当我们用Asp.net技术开发Web应用程序时，我们可以多从网络协议的方面去思考我们的应用程序，而不是只是单单停留在对服务器控件的拖拉的使用，这样也可以帮助我们开发一个自己的自定义web服务器。在这里我想同时把我对Asp.net的本质的理解和大家分享下，如果有什么不对的地方，还请大家指出，首先，当我们设计一个算法的时候要明确输入参数和算法的返回（算法也就是也就是一个处理程序），其实Asp.net开发的web网页可以理解为一个处理程序，因为我们在web浏览器中所看到的都是HTML文档（HTML也就是Asp.net网页处理后程序的输出,即算法的返回），然而输入参数也就是用户通过浏览器输入的一个Http请求（可以说是请求的一个URI地址），asp.net这门技术就帮助我们把请求的aspx页面翻译为HTML文档，然后HTML文档通过HTTP协议把HTML文档发送给浏览器，浏览器再把这么标签（HTML文档只是一串字符串，如果没有浏览器的解析我们看到的也是一些字符串，而不是可视化的界面了）解析为可视化的界面。这样一次web请求也就结束。后面也会和大家分享下Asp.net中背后替我们所做事情的一些对象，这里还是回到Http协议的介绍吧。

一、**HTTP**协议的简介

HTTP中文为超文本传输协议，从名字上很容易理解，Http协议就是将超文本标记语言的文档（即Html文档）从web服务传送到客户端的浏览器。它属于一个应用层的协议。

二、网络的工作过程

当用户要访问网络中的某个网页时，大致要经过以下几个步骤：

1. 用户首先要确定网页文件所在的URL（统一资源定位符，也就是网页在网络上的家庭住址，通过这个地址就可以找到这个网页）如www.cnblogs.com
2. 浏览器向DNS(域名服务器)发出请求，告诉DNS说："我要把www.cnblogs.com转化为它所定义的IP地址",这里可以简单把DNS理解为一个字典，知道域名就可以知道域名对于的IP地址，他们有这个一个映射的关系
3. DNS收到请求后就开始查询，查到后向浏览器返回结果。如域名为www.cnblogs.com对应的IP地址为61.155.169.116
4. 知道IP地址后，浏览器向IP地址为61.155.169.116的主机发出与端口号80建议一条TCP连接请求（HTTP协议是建立在传输层TCP的基础上的），80端口是服务器提供web服务的默认端口
5. 建立连接后，浏览器发出一条HTTP请求，如 GET http://www.cnblogs.com/ HTTP/1.1
6. 当域名为www.cnblogs.com的服务器接受到请求后，向浏览器发送一个html文件
7. 文件发送完后，由服务器主动关闭TCP连接。
8. 浏览器接收传送来的页面并显示
9. 如果Html文件中包含图片，还要与服务器再次建立一个TCP连接，以便可以下

载图片

上面介绍的步骤中，浏览器发出一个请求后，如何把一个服务器上的HTML文档下载到请求网页的主机上呢？这个过程就是由HTTP完成，即完成超文本文件的传送，HTTP协议是web服务器的基础。

二、HTTP请求

Http请求由三部分组成：请求行、请求头和请求数据，一个HTTP请求的格式一般如下：

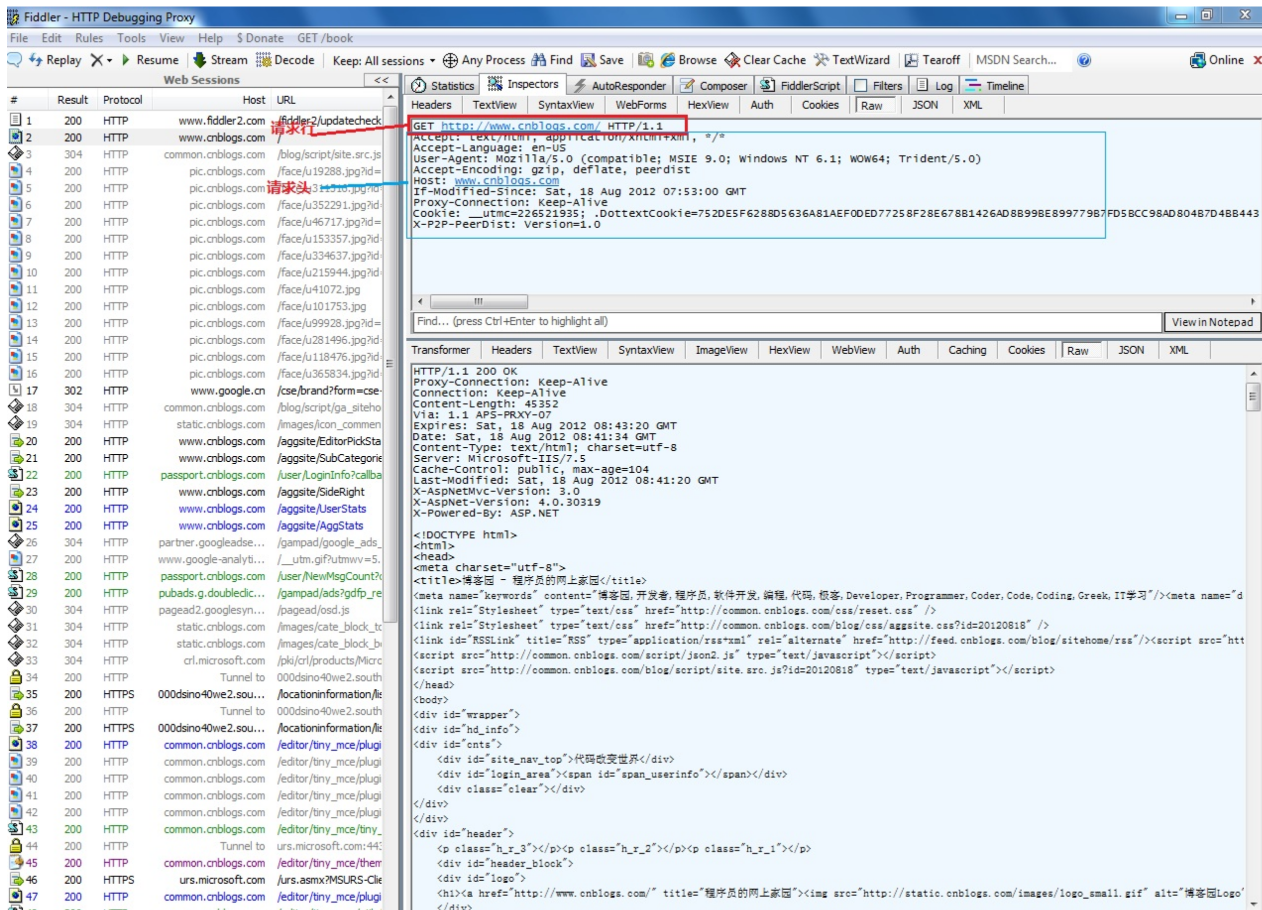
```
请求方法 URL HTTP版本号
请求头信息
 <一个空行>
请求数据
```

HTTP请求的方法如下表：

方法	描述
Get	返回URL所指的文档，一般用来请求下载Web网页
Head	请求文档头，它类似Get方法，只是Web服务器返回指定文档的首部信息
Post	它与Get方法相反，请求服务器接受指定文档，但它不替换已有的文档，只是将新数据附加在它的后面
Put	它与Get方法类似，用从客户端传送的数据取代指定文档中的内容，使客户可以向远程Web服务器传送网页等文件
Delete	请求服务器删除指定的页面
Options	允许客户端查看服务器的性能
Trace	用于测试允许客户端查看的消息回收过程

经常使用的是Get和Post方法，当使用Get方法发出请求时，请求数据为空，所以此时的HTTP请求行就由两部分组成：请求行和请求头信息，下面我们形象看看具体的HttP的实例：

当在浏览器中地址栏里面输入：www.cnblogs.com，此时我们相当于发出一个HTTP请求，具体为：



并且从图中可以看出网页中含有图片脚本等文件时，客户端会继续与服务器发出请求，请求所需要的图片和脚本文件。

补充：经一位朋友的留言中，在这里我补充下，现在通常是只建立一个TCP连接，通过HTPP 请求头的Connetion字段来指明，当服务器收到附带有Connection: Keep-Alive的请求时，它也会在响应头中添加一个同样的字段来使用Keep-Alive。这样一来，客户端和服务器之间的HTTP连接就会被保持，不会断开，（一些特殊情况除外）当客户端发送另外一个请求时，就使用这条已经建立的连接。

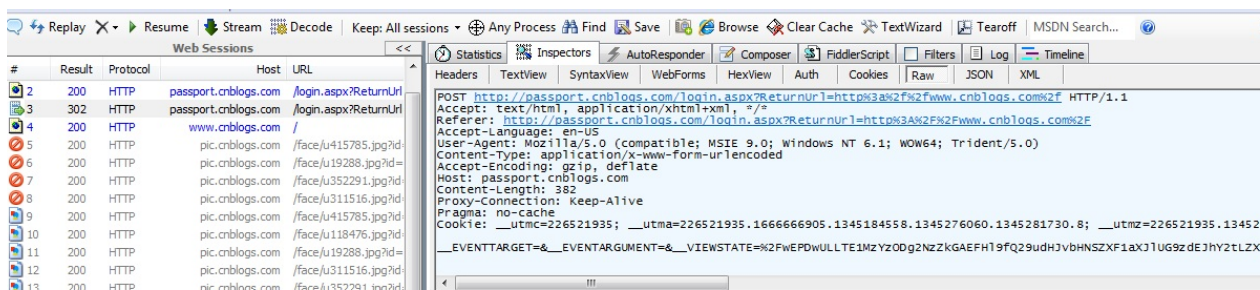
下面介绍下请求头的信息：

Accept:表示客户端接收的数据类型。例如，Accept：text/html表示客户端可接收HTML类型的文本

User Agent:表示客户端软件类型

Referer:表示的是上一连接的url，如跳转到本页面的上一页面url。

上图是一个通过Get方法把一个HTML文件下载到本例浏览器中显示的过程，当我们在博客园主页面点登陆后输入用户名和密码后点确认按钮后，此时我们发出的HTTP的请求是通过Post方法，下面是一个截图：



从图中可以看出，通过Post方法发出的HTTP请求中有一个空行（空行后为请求数据），而Get方法发出的请求中没有。

三、**HTTP**响应

同样，Http响应也是由三部分组成：状态行，响应头和响应数据组成，Http响应格式如下：

```
状态行
响应头
<一个空行>
响应数据
```

状态行以HTTP版本号开始，后面跟着3为数字，代表响应代码，响应代码用来告诉客户端,服务器是否产生了预期的响应。如HTTP/1.1 200 OK.

HTTP/1.1中定义五种响应代码：

1xx：指示信息--表示请求已接收，继续处理 2xx：成功--表示请求已被成功接收、理解、接受 3xx：重定向--要完成请求必须进行更进一步的操作 4xx：客户端错误--请求有语法错误或请求无法实现 5xx：服务器端错误--服务器未能实现合法的请求

具体响应代码的说明见下：

响应	说明
成功响应	
200	OK,请求成功
201	OK，建立新的资源
202	请求被接受，处理未完成
204	OK,但没有内容返回
重定向，需要用户代理执行更多的工作	
301	所请求的资源已被指派为新的固定 URL
302	所请求的资源临时位于另外的 URL
304	文件没有修改
客户端错误	
400	错误的请求
401	未被授权：该请求要求用户认证
402	不明原因的禁止
404	没有找到
服务器端错误	
500	内部服务器错误
501	没有实现
502	错误的网关
504	服务器暂时失效

HTTP响应头用于服务器向客户端提供请求文档信息或服务端的状态信息，如图：

响应头	说明
Server	Web 服务器程序的信息
Data	当前服务器的日期和时间
Last-Modified	请求文档最近一次修改的时间
Expires	请求文档过期时间
Content-length	数据长度（字节）
Content-type	数据 MIME 类型
Content-Encoding	文档编码方式

四、总结

到这里这篇文章也算是说完了，HTTP协议只是应用层中协议的其中之一，应用层还有其他的一些协议，比如FTP（文件传输协议），SMTP(电子邮件协议)等，这些协议在后面都会有所介绍。后面一个专题打算应用HTTP协议的只是自定义一个简单的Web服务器来模拟我们平常在浏览器中输入网址后发送Http请求和服务器返回响应的过程。

[C# 网络编程系列]专题三：自定义Web服务器

前言：

经过前面的专题中对网络层协议和HTTP协议的简单介绍相信大家对网络中的协议有了大致的了解的， 本专题将针对HTTP协议定义一个Web服务器，我们平常浏览网页通过在浏览器中输入一个网址就可以看到我们想要的网页，这个过程中浏览器只是一个客户端，浏览器（应用层应用程序）通过HTTP协议把用户请求发送到服务端， 服务器接受到发送来的HTTP请求，然后对请求进行处理和响应，最后把响应的内容发送给客户端（浏览器这里充当了用户代理的客户端），浏览器再对接受到的响应内容（一般是HTML文件）进行解释并且显示出来。这就是一次完整的用户请求/响应模型，本专题所讲述的是一个简单的Web服务器，其他一些大型的Web服务器（IIS，Apache）也是这样的一个原理， 本专题只是简单讲述Web服务器的实现原理。

一、Socket编程实现一个简单的Web服务器

Socket这个概念是在Unix系统中提出来的。在Unix的时代，为了解决传输层的编程问题，Unix提供了类似于文件操作的网络操作方式——Socket,通过Socket，我们就可以像操作文件一样通过打开、写入、读取、关闭等操作完成网络编程，这样就使得网络编程可以统一到文件操作方面，这样就使我们更容易地编写网络应用程序。需要注意的是，应用层的协议需要网络程序专门处理，Socket不负责应用层协议，仅仅负责传输层的协议。

现在介绍下网络端口号（port）的概念，在同一个网络地址中，为了区分使用相同协议的不同应用程序，为不同的应用程序分配一个数字编号，我们把这个编号就成为网络端口号（就是区分同一个网络地址中不同的进程）。端口号是由一个两个字节的整数，所以取值范围为0~65535，这些端口号又分为三类：

1. 第一类的范围是0~1023，称为众所周知的端口，这些端口号由特定的网络程序使用，例如，TCP协议使用80端口来完成Http协议的传输。
2. 第二类的范围是1024~49151，称为登记端口，一般情况下不应该在程序中使用。
3. 第三类的范围是49152~65535，称为私有端口， 这些端口可以由普通用户程序使用。

在我们用Socket开发网络应用程序中，还有一个就是端点的概念，在网络中，通过IP地址，协议和端口号可以唯一地确定网络上的一个应用程序，其中把IP地址和端口的组合叫做端点（EndPoint）。每个Socket需要绑定到一个端点上与其他端点进行通信。

介绍完基本的一些概念后，下面演示通过Socket编程实现一个简单的Web服务器，此实例中就是简单向浏览器返回一个固定的静态页面,实现代码如下：

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;
```

```
namespace WebServer
{
    /// <summary>
    /// 实现一个简单的Web服务器
    /// 该服务器向请求的浏览器返回一个静态的HTML页面
    /// </summary>
    class Program
    {
        static void Main(string[] args)
        {
            // 获得本机的Ip地址，即127.0.0.1
            IPAddress localaddress =IPAddress.Loopback;

            // 创建可以访问的断点，49155表示端口号，如果这里设置为0，表示使
            IPEndPoint endpoint = new IPEndPoint(localaddress,4915

            // 创建Socket对象,使用IPv4地址，数据通信类型为数据流，传输控制
            Socket socket = new Socket(AddressFamily.InterNetwork,S

            //将Socket绑定到断点上
            socket.Bind(endpoint);
            // 设置连接队列的长度
            socket.Listen(10);

            while (true)
            {
                Console.WriteLine("Wait an connect Request...");
                // 开始监听，这个方法会堵塞线程的执行，直到接受到一个客户端的
                Socket clientsocket =socket.Accept();

                // 输出客户端的地址
                Console.WriteLine("Client Address is: {0}", client
                // 把客户端的请求数据读入保存到一个数组中
                byte[] buffer =new byte[2048];

                int receivelength = clientsocket.Receive(buffer, 2
                string requeststring = Encoding.UTF8.GetString(buf

                // 在服务器端输出请求的消息
                Console.WriteLine(requeststring);

                // 服务器端做出相应内容
                // 响应的状态行
                string statusLine ="HTTP/1.1 200 OK\r\n";
                byte[] responseStatusLineBytes = Encoding.UTF8.Get
                string responseBody = "<html><head><title>Default 
                string responseHeader =
                    string.Format(
                        "Content-Type: text/html; charset=UTf-8\r\

                byte[] responseHeaderBytes = Encoding.UTF8.GetByte
                byte[] responseBodyBytes = Encoding.UTF8.GetBytes(
```

```
                    // 向客户端发送状态行
                    clientsocket.Send(responseStatusLineBytes);

                    // 向客户端发送回应头信息
                    clientsocket.Send(responseHeaderBytes);

                    // 发送头部和内容的空行
                    clientsocket.Send(new byte[] { 13, 10 });

                    // 想客户端发送主体部分
                    clientsocket.Send(responseBodyBytes);

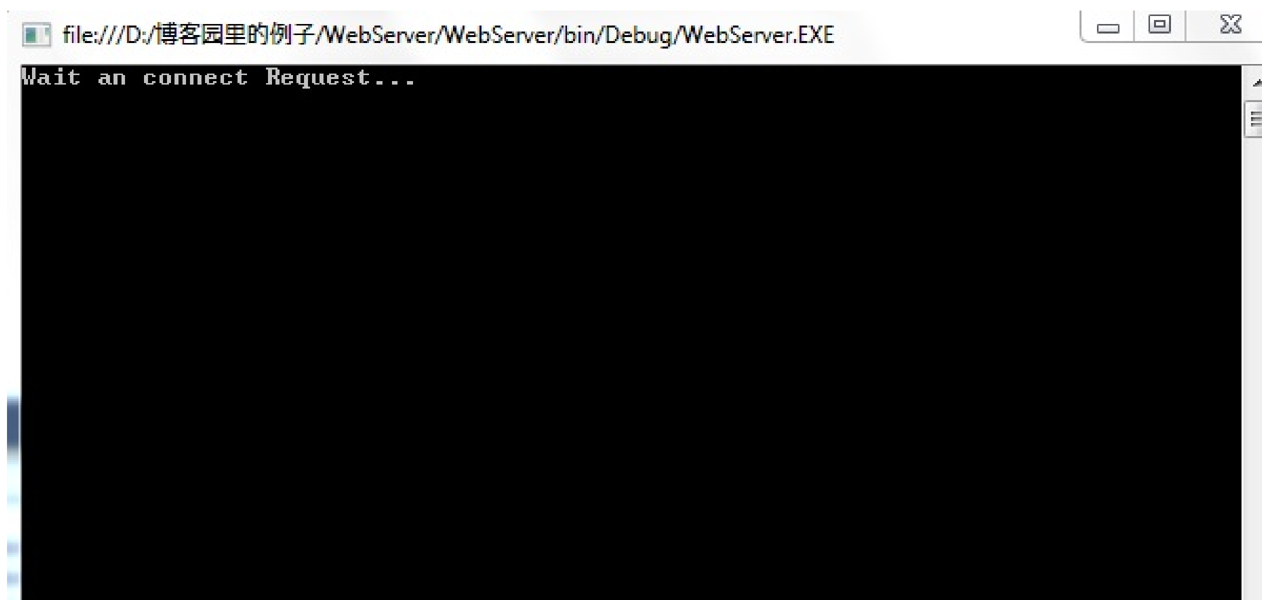
                    // 断开连接
                    clientsocket.Close();
                    Console.ReadKey();
                    break;
                }

                // 关闭服务器
                socket.Close();

            }
        }
    }
```

运行结果：

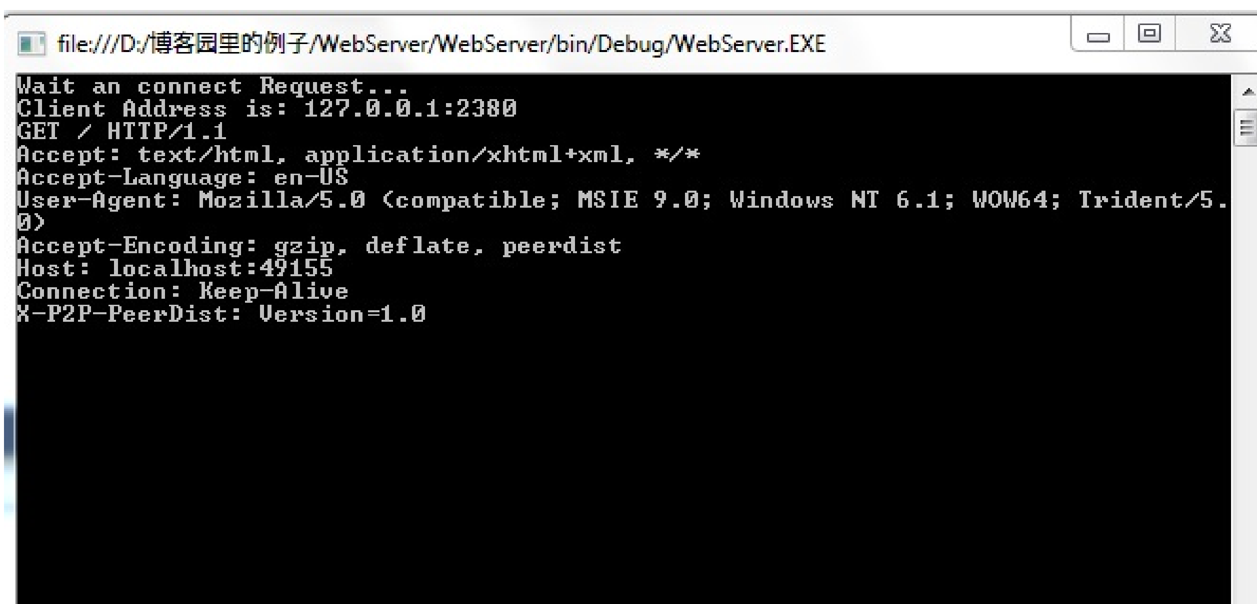
首先运行服务端后的界面：



在浏览器中输入http://localhost:49155/ 则浏览器可以看到如下的所示的结果：



此时在服务器端显示的输出为：



这里只是简单实现了一个web服务器的功能，当然实际的Web服务器通过用户的发来的Http请求中获得请求文件类型，请求文件名以及请求目录等信息，然后Web服务器根据这些请求信息从服务器的物理目录中寻找请求的文件，如果在服务器中找到请求的文件，然后服务器把响应内容发送给客户端。这里只是通过这个简单的Web服务器让大家理解请求/响应模型以及Web服务器的工作原理，一些复杂的Web服务器也是在此基础进行一些其他功能的扩展。

二、基于TcpListener的Web服务器

在.net平台下， 为了简化网络编程，.net对套接字又进行了一次封装，封装后的类是在System.Net.Sockets命名空间下的**TcpListener**类和**TcpClient**类， 使用**TcpListener**类用来监听和接收传入的连接请求，在该类的构造函数中只需要传递一组网络端点信息就可以准备好监听参数，而不需要设置使用的网络协议等细节，调用Start方法后，监听工作就开始（间接调用了Socket.Listen方法）， AcceptTcpClient方法将阻塞进程，直到一个客户端发来连接请求为止，这个方法返回一个

封装了Socket的**TcpClient**对象，同时从传入的连接队列中删除该客户端的连接请求。此时通过这个TcpClient对象与客户端进行通信。

下面是基于TcpListener和TcpClient的一个简单的Web服务器的代码：

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

namespace TcpWebserver
{
    class Program
    {
        static void Main(string[] args)
        {
            // 获得本机的Ip地址，即127.0.0.1
            IPAddress localaddress =IPAddress.Loopback;

            // 创建可以访问的断点，49155表示端口号，如果这里设置为0，表示使
            IPEndPoint endpoint = new IPEndPoint(localaddress, 4915

            // 创建Tcp 监听器
            TcpListener tcpListener = new TcpListener(endpoint);

            // 启动监听
            tcpListener.Start();
            Console.WriteLine("Wait an connect Request...");
            while (true)
            {
                // 等待客户连接
                TcpClient client =tcpListener.AcceptTcpClient();
                if (client.Connected == true)
                {
                    // 输出已经建立连接
                    Console.WriteLine("Created connection");
                }

                // 获得一个网络流对象
                // 该网络流对象封装了Socket的输入和输出操作
                // 此时通过对网络流对象进行写入来返回响应消息
                // 通过对网络流对象进行读取来获得请求消息
                NetworkStream netstream = client.GetStream();
                // 把客户端的请求数据读入保存到一个数组中
                byte[] buffer = new byte[2048];

                int receivelength = netstream.Read(buffer, 0, 2048
                string requeststring = Encoding.UTF8.GetString(buf

                // 在服务器端输出请求的消息
                Console.WriteLine(requeststring);
```

```
                // 服务器端做出相应内容
                // 响应的状态行
                string statusLine = "HTTP/1.1 200 OK\r\n";
                byte[] responseStatusLineBytes = Encoding.UTF8.GetB
                string responseBody = "<html><head><title>Default F
                string responseHeader =
                    string.Format(
                        "Content-Type: text/html; charset=UTf-8\r\n

                byte[] responseHeaderBytes = Encoding.UTF8.GetBytes
                byte[] responseBodyBytes = Encoding.UTF8.GetBytes(r

                // 写入状态行信息
                netstream.Write(responseStatusLineBytes, 0, respons
                // 写入回应的头部
                netstream.Write(responseHeaderBytes, 0, responseHea
                // 写入回应头部和内容之间的空行
                netstream.Write(new byte[] { 13, 10 }, 0, 2);

                // 写入回应的内容
                netstream.Write(responseBodyBytes, 0, responseBodyB

                // 关闭与客户端的连接
                client.Close();
                Console.ReadKey();
                break;
            }

            // 关闭服务器
            tcpListener.Stop();
        }
    }
}
```

程序的输出结果和前面的用Socket实现的效果相同，这里就不再贴图了，这里实现的Web服务器都是建立控制台的应用程序来实现的，感兴趣的朋友也可以用Windows窗体进行实现，同时这里也只是简单列出了采用同步的方式进行实现的，同时**TcpListener**类和**TcpClient**类同时支持异步操作的方法，下面列出这个两个类中异步操作的方法如下表：

类	方法	说明
TcpListener	BeginAcceptTcpClient	开始一个异步操作接受一个传入的连接
EndAcceptTcpClient	异步接受传入的连接，并创建新的TcpClient对象来处理客户端的通信	
TcpClient	BeginConnect	开始一个对远程主机连接的异步请求
EndConnect	异步接受传入的连接尝试。	

如果想了解线程同步和异步的朋友可以参考我的多线程处理系列：http://www.cnblogs.com/zhili/archive/2012/07/21/ThreadsSynchronous.html

三、总结

到这里这篇文章就差不多介绍到这里了，本专题是介绍如何自定义一个简单Web服务器，通过这个专题希望大家可以对Web服务器的工作过程有一个简单的了解。

另外在这个专题里面我们是用IE浏览器进行发送客户请求的，所以后面专题将介绍自定义一个浏览器，通过我们自定义的浏览器来对Web服务器发送请求，然后在自己自定义的浏览器中把响应消息显示出来。

[C# 网络编程系列]专题四：自定义Web浏览器

前言：

前一个专题介绍了自定义的Web服务器，然而向Web服务器发出请求的正是本专题要介绍的Web浏览器，本专题通过简单自定义一个Web浏览器来简单介绍浏览器的工作原理，以及帮助一些初学者揭开浏览器这层神秘的面纱（以前总感觉这些应用感觉很深奥的，没想到自己也可以自定义一个浏览器出来），下面不啰嗦了，进入正题。

一、**Web**浏览器的介绍

Web浏览器是指可以显示Web服务器或者本地文件系统中的Html文件内容，并让用户与这些文件交互的一种软件，它是网络服务的客户端浏览程序，可向Web服务器发送请求，并对服务器返回的超文本信息和各种媒体、图片进行解释和显示。

浏览器主要通过Http协议与服务器交互并获得网页，现在主流的浏览器有：IE，Google Chrome(谷歌浏览器)、Mozilla Firefox（火狐）、Opera浏览器、世界之窗、360安全浏览器等。

Web浏览器的组成

一般来说，Web浏览器由控制器和解释器组成，控制器负责解释鼠标点击与键盘输入，并调用其他组件用于执行用户的指定的操作。例如，当用户输入一个URL或单击一个超链接时，控制器接收并分析该命令，调用一个HTML解释器来解释该页面，并将解释后的结果显示在用户的浏览器上。

解释器对于浏览器来说是很重要的， 解释器，也就是解释引擎，负责对网页语法（如HTML、Javascript）的解释并显示网页，解释器决定了浏览器如何显示页面，是浏览器最重要最核心的一个部分，所以一般我们所说的浏览器内核指的就是浏览器的解释器。

不同浏览器产品可能使用同一个内核，浏览器内核常见的有四种：**Trident**、**Gecko**、**Presto**和**Webkit,** 他们与主流浏览器的对于关系如下表：

内核	浏览器产品
Trident	IE,Maxthon(傲游)，世界之窗，腾讯TT,搜狗浏览器，360安全浏览器
Gecko	Mozilla Firefox(火狐)
Presto	Opera浏览器
Webkit	苹果Safari浏览器，Google Chrome(谷歌浏览器)及苹果Iphone手机浏览引擎

二、**.NET**平台对浏览器开发的支持

浏览器软件一般都不是从头开始开发的，而是基于某种内核之上的扩展。同样，微软.NET平台封装了IE浏览器内核并以COM组件的形式提供用户，这个COM组件就是**WebBrowser**控件，该控件实现了浏览器中几乎全部的基本功能。

WebBrowser就是一个以IE(Trident)为内核，实现了基本功能的Web浏览器。使用WebBrowser控件可以在Windows窗体应用程序中浏览网页，WebBrowser控件位于工具箱中，使用时只需要将它直接拖拉到程序窗口中。

下面介绍WebBrowser控件的常用的属性和方法

这里我直接摘自MSDN中的一个表来说明的：

名称	说明
Document 属性	获取一个对象，用于提供对当前网页的 HTML 文档对象模型 (DOM) 的托管访问。
DocumentCompleted 事件	网页完成加载时发生。
DocumentText 属性	获取或设置当前网页的 HTML 内容。
DocumentTitle 属性	获取当前网页的标题。
GoBack 方法	定位到历史记录中的上一页。
GoForward 方法	定位到历史记录中的下一页。
Navigate 方法	定位到指定的 URL。
Navigating 事件	导航开始之前发生，使操作可以被取消。
ObjectForScripting 属性	获取或设置网页脚本代码可以用来与应用程序进行通信的对象。
Print 方法	打印当前的网页。
Refresh 方法	重新加载当前的网页。
Stop 方法	暂停当前的导航，停止动态页元素，如声音和动画。
Url 属性	获取或设置当前网页的 URL。设置该属性时，会将该控件定位到新的 URL。

三、在**.NET**平台下自定义**Web**浏览器

下面是自定义浏览器的一些效果图:

浏览器的主页面：



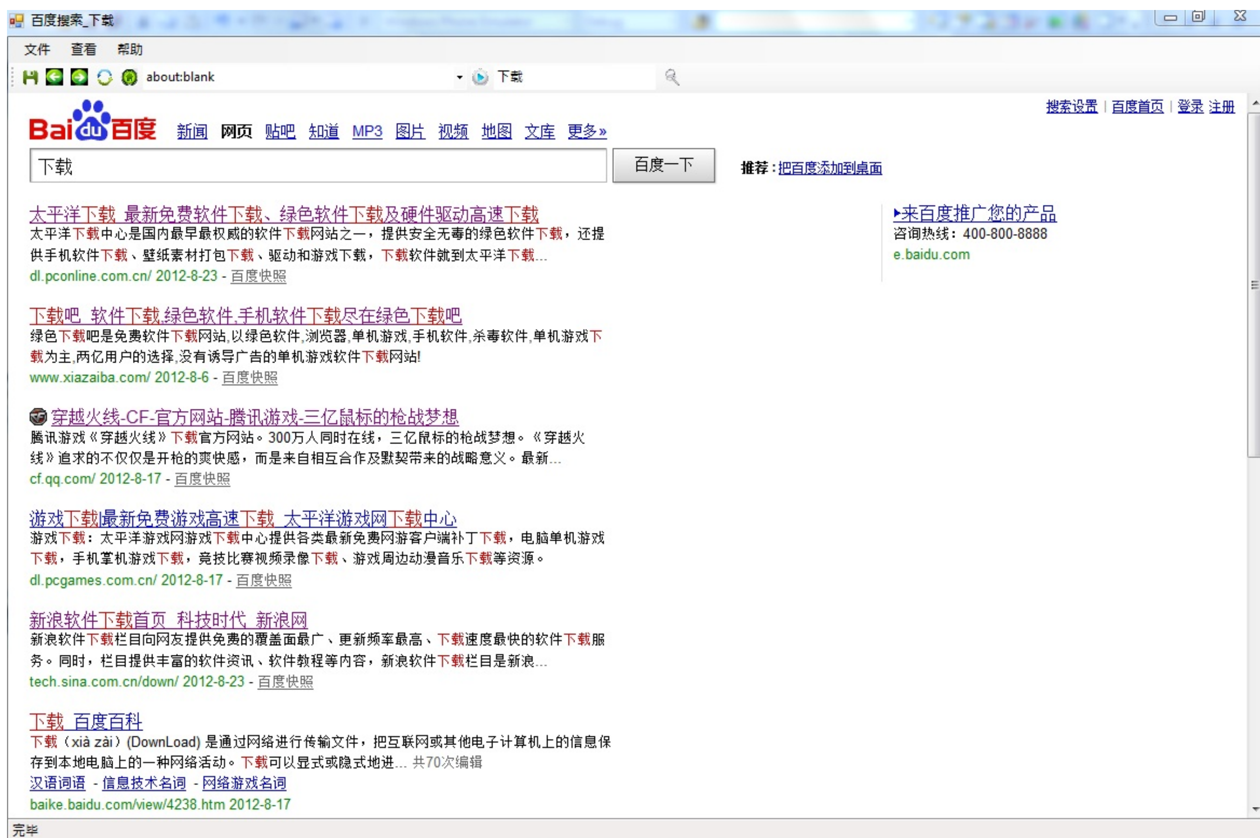
点击查看->源文件->UTF-8后就可以查看Html的源码界面：



关于窗口的设计页面：



在搜索栏里面输入下载后利用百度搜索引擎后显示的页面：



四、总结

本专题主要对Web浏览器的介绍，并且自定义了一个简单的Web浏览器，希望通过本专题，大家可以对浏览器的工作原理有所了解。如果大家有什么任何疑问或者我有说的不对的地方还请大家留言来告诉我。讲到这里本专题也算结束，后面将介绍TCP编程和UDP编程，以及介绍完这两个专题后将为大家介绍如何开发一个即时通信聊天的工具（类似QQ的应用程序）。

补充： 鉴于很多朋友推荐使用非IE内核来实现一个浏览器的功能，这里分享下Webkit.net(WebKit .NET 是一个 C# 的组件封装了 WebKit 浏览器引擎，通过它可以在 .NET 应用中简单的使用(Google Chrome的内核)WebKit 浏览器引擎）的源码地址，也给有兴趣的朋友研究，当然我也会研究下，之后会和大家分享下这个工具的使用。同时感谢大家的留言和建议。 Webkit.net源码地址为：http://sourceforge.net/projects/webkitdotnet/

如果觉得有帮助的还请大家推荐下， 源代码链接为：http://files.cnblogs.com/zhili/WebBrowser.zip

[C# 网络编程系列]专题五：TCP编程

前言

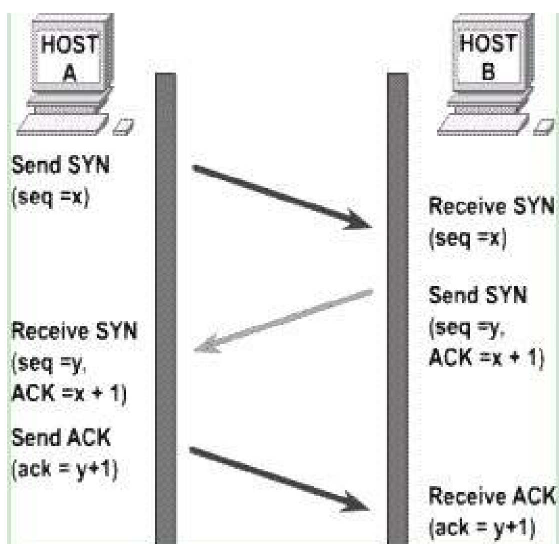
前面专题的例子都是基于应用层上的HTTP协议的介绍， 现在本专题来介绍下传输层协议——TCP协议，主要介绍下TCP协议的工作过程和基于TCP协议的一个简单的通信程序，下面就开始本专题的正文了。

一、**TCP**的工作过程

首先TCP是一种面向连接的，可靠的，基于字节流的传输层通信协议。TCP的工作过程可以分为三个阶段：一、连接的建立； 二、传输数据； 三、断开连接，下面就对这三个过程分别介绍下：

1.1 连接的建立

TCP的连接建立就像打电话一样， 我们打电话时，拨一个号码的号码并不是立即就可以接通的，期间会有一个"嘟 嘟"的呼叫过程， 这就好比是TCP协议的连接的建立阶段。当我们用TCP编写的程序，必须先建立TCP连接。TCP协议的连接建立通过三次握手来完成的，下面是在网上找的一张TCP三次握手的图片：



下面就对这三次握手简单的介绍：

第一次握手：建立连接时，客户端发送SYN包（seq=x）到服务器，并进入SYN_Send状态，等待服务器确认

第二次握手：服务器收到SYN包，必须确认客户的SYN（ACK=x+1），同时自己也发送一个SYN包（SEQ=y），即SYN+ACK包，此时服务器进入SYN_Recv状态

第三次握手：客户端收到服务器的SYN+ACK包，向服务器发送确认包ACK（ACK=y+1）,此包发送完毕，客户端和服务器进入Established(建立)状态，完成三次握手。

简单理解三次握手就是发送一个检验包给对方然后互相确认，双方都接到确认的一个信号时，这时候双方就建立了连接（就像我们打电话时，如果没人说话时就会说下 "喂"，说这句"喂" 也就是希望得到对方的一个确认，虽然这里双方已经建立了连接的，这里只是更形象的说明下三次握手的过程）。

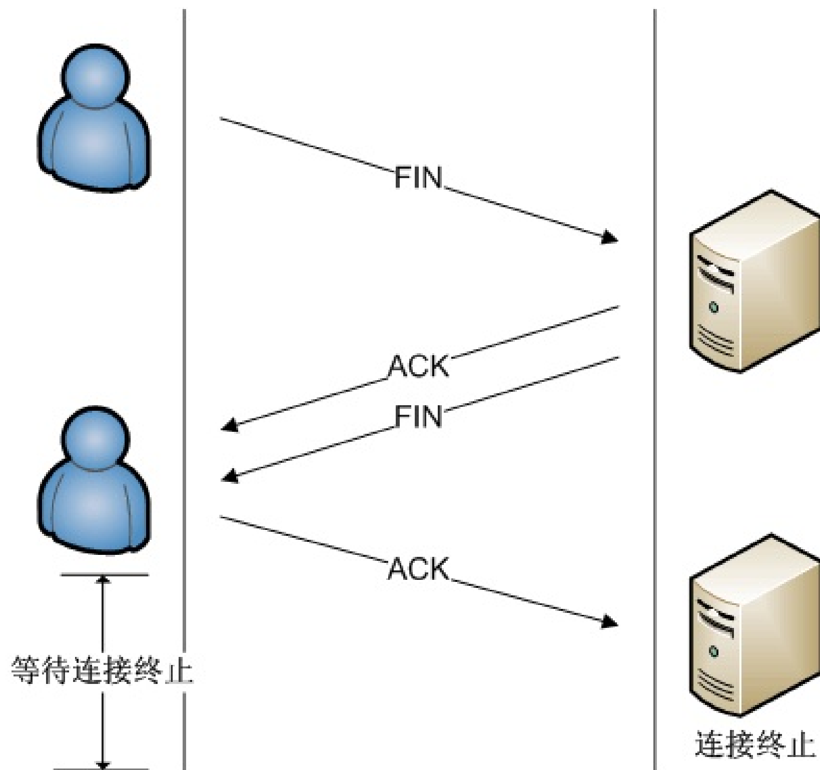
1.2 传输数据

双方建立了连接，即在双方建立了一个通信通道（就像一座桥一样，在两端建立了一个通路，用桥来比喻通信通道主要是因为最近有一则新闻：哈尔滨阳明滩大桥坍塌事件），建立连接之后，当然是传输我们需要传输的数据到对方的，这里就开始简单介绍下传输数据的过程。

利用TCP传输数据时，数据是以字节流的形式进行传输，客户端与服务器端建立连接后，发送方需要先将发送的数据转换为字节流，然后将其发送给对方，发送数据时，可以通过程序不断地将数据流陆续写入TCP的发送缓冲中，然后TCP自动从发送缓冲中提取一定量的数据，将其组成TCP报文段发送到IP层，再通过IP层（也就是网络层）之下的网络接口发送出去；接受端从IP层接收到TCP报文段后，将其暂时保存在接受缓冲中，然后我们通过程序依次读取接受缓冲中的数据，从而达到相互通信的目的（简单的说就发送方把数据转换为数据流，再把数据流存储在发送缓冲中，然后传输层低层的协议从发送缓冲中读取数据把数据发送出去，然后接收端从底层接受到数据把数据存储在接收端的缓冲中，然后我们写的程序只是从缓冲中依次读取数据，然后显示出来，在客户端我们写代码做的事情是把数据写入Write写入发送端的缓冲中，然后服务器端（接收端）用Read方法在自己的缓冲中读取数据，用一句话概括，TCP的传输就是对数据的写——读操作）括号中的内容只是我个人理解，因为这样我感觉理解起来比较容易，对于刚开始接触TCP的朋友可以这样理解，然后再一句句话去扩展。

1.3 断开连接

发送完数据之后，最后就是断开连接了，下面是网上断开的连接的一张图片（断开一个连接需要经过四次握手）：



TCP的工作过程就分为上面三个过程，TCP编程是作为上层应用编程的基础，就像之前专题中基于HTTP协议的Web服务器，Web浏览器，其传输层都用的是TCP协议进行传输的，还有基于FTP（文件传输协议），IMAP（交互式邮件存取协议）POP3（邮局协议的第3个版本） 和SMTP（简单邮件传输协议）的网络应用其传输层都用的是TCP协议，而不是UDP等其他传输层协议。

二、基于**TCP**协议的简单通信程序

这里简单实现了一个客户端与服务器间的通信程序，核心代码为：

客户端连接服务器端代码：

```
private void btnConnect_Click(object sender, EventArgs e)
        {
            // 通过一个线程发起请求,多线程
            Thread connectThread = new Thread(ConnectToServer);
            connectThread.Start();
        }

        // 连接服务器方法,建立连接的过程
        private void ConnectToServer()
        {
            try
            {
                // 调用委托
                statusStripInfo.Invoke(showStatusCallBack, "正在连接
                if (tbxserverIp.Text == string.Empty || tbxPort.Tex
                {
                    MessageBox.Show("请先输入服务器的IP地址和端口号");
                }

                IPAddress ipaddress = IPAddress.Parse(tbxserverIp.T
                tcpClient = new TcpClient();
                tcpClient.Connect(ipaddress, int.Parse(tbxPort.Text

                // 延时操作
                Thread.Sleep(1000);
                if (tcpClient != null)
                {
                    statusStripInfo.Invoke(showStatusCallBack, "连接
                    networkStream = tcpClient.GetStream();
                    reader = new BinaryReader(networkStream);
                    writer =new BinaryWriter(networkStream);
                }

            }
            catch
            {
                statusStripInfo.Invoke(showStatusCallBack,"连接失败"
                Thread.Sleep(1000);
                statusStripInfo.Invoke(showStatusCallBack,"就绪");
            }
        }
```

客户端发送消息的代码：

```
// 发送消息
        private void btnSend_Click(object sender, EventArgs e)
        {
            Thread sendThread = new Thread(SendMessage);
            sendThread.Start(tbxMessage.Text);
        }

        private void SendMessage(object state)
        {
            statusStripInfo.Invoke(showStatusCallBack, "正在发送..."
            try
            {
                writer.Write(state.ToString());
                Thread.Sleep(5000);
                writer.Flush();
                statusStripInfo.Invoke(showStatusCallBack, "完毕");

                tbxMessage.Invoke(resetMessageCallBack, null);
                lstbxMessageView.Invoke(showMessageCallback, state
            }
            catch
            {
                if (reader != null)
                {
                    reader.Close();
                }
                if (writer != null)
                {
                    writer.Close();
                }
                if (tcpClient != null)
                {
                    tcpClient.Close();
                }

                statusStripInfo.Invoke(showStatusCallBack, "断开了连
            }
        }
```

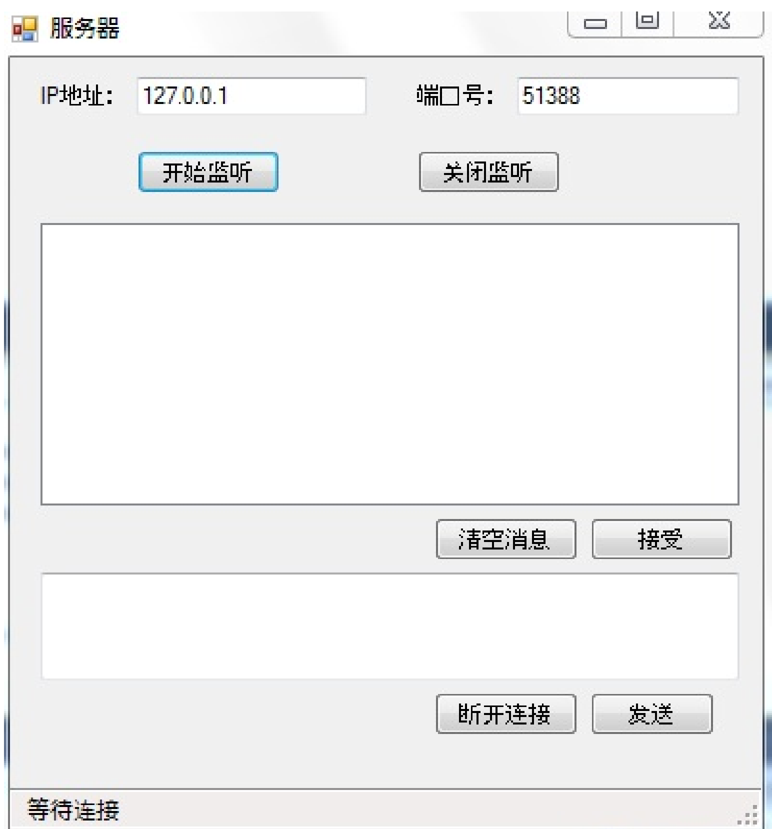
服务器端接受开始监听客户端请求的代码：

```
// 开始监听
        private void btnStart_Click(object sender, EventArgs e)
        {

            tcpLister = new TcpListener(ipaddress,Port);
            tcpLister.Start();
            // 启动一个线程来接受请求
            Thread acceptThread =new Thread(acceptClientConnect);
            acceptThread.Start();
        }

        // 接受请求
        private void acceptClientConnect()
        {
            statusStripInfo.Invoke(showStatusCallBack,"正在监听");
            Thread.Sleep(1000);
            try
            {
                statusStripInfo.Invoke(showStatusCallBack,"等待连接"
                tcpClient = tcpLister.AcceptTcpClient();
                if (tcpLister != null)
                {
                    statusStripInfo.Invoke(showStatusCallBack,"接受
                    networkStream = tcpClient.GetStream();
                    reader = new BinaryReader(networkStream);
                    writer = new BinaryWriter(networkStream);
                }
            }
            catch
            {
                statusStripInfo.Invoke(showStatusCallBack, "停止监听
                Thread.Sleep(1000);
                statusStripInfo.Invoke(showStatusCallBack, "就绪");
            }
        }
```

现在看看运行的结果：首先先启动服务器然后点开始监听，此时线程会堵塞，直到接受到一个连接请求位置



然后运行客户端，在IP地址和端口处输入服务器端的IP地址和端口号，点击连接服务器按钮后的界面如下：



通过接受按钮和发送按钮来实现双方的通信，实现界面如下：



三、总结

到这里本专题的内容将的差不多了， 本专题主要介绍了基于TCP协议工作过程和在net平台下自定义了一个简单通信的程序，希望本专题可以给那些初次接触TCP协议的朋友一些帮助，(大牛们应该直接可以闪过的)，在后面的专题我将和大家分享UDP编程，讲完UDP编程后将结合这两章的内容实现一个类似QQ的即时聊天的工具，希望这些对大家有帮助，如果大家有任何问题和有感兴趣的专题需要了解的，可以给我留言，在之后的文章都会和大家来分享。

觉得看了后有帮助的朋友麻烦推荐下，也给我继续下去的动力，如果大家有什么感兴趣的专题也可以留言告诉我，我会通过学习后也会相继和大家分享。

下面是本程序源代码：

http://files.cnblogs.com/zhili/%E7%AE%80%E5%8D%95%E9%80%9A%E4%BF%A1%E7%A8%8B%E5%BA%8F.zip

[C# 网络编程系列]专题六：UDP编程

引用：

前一个专题简单介绍了TCP编程的一些知识，UDP与TCP地位相当的另一个传输层协议，它也是当下流行的很多主流网络应用（例如QQ、MSN和Skype等一些即时通信软件传输层都是应用UDP协议的）底层的传输基础，所以在本专题中就简单介绍下UDP的工作原理和UDP编程的只是，希望可以对刚接触网络编程的朋友起到入门的作用。

一、UDP介绍

UDP和TCP都是构建在IP层之上传输层的协议，但UDP是一种简单、面向数据报**(Sock_Dgram)**的无连接协议，提供的是不一定可靠的传输服务。

然而TCP是一种面向连接、可靠的，面向字节流（**Sock_Stream**）的传输协议,对于"无连接"是指在正式通信前不必与对方先建立连接，不管对方状态如何都可以直接发送过去(就如QQ中通过QQ号查看好友后发送添加好友请求，此间不需要考虑对方的状态如何，都照样发送请求)。从UDP和TCP的定义中就可以看出它们两者的区别了，（1）UDP的可靠性不如TCP，因为TCP传输前要首先建立连接，这样就增加了TCP传输的可靠性，所以UDP也被称为不可靠的传输协议，关于TCP的介绍可以看我上一篇博客的介绍。

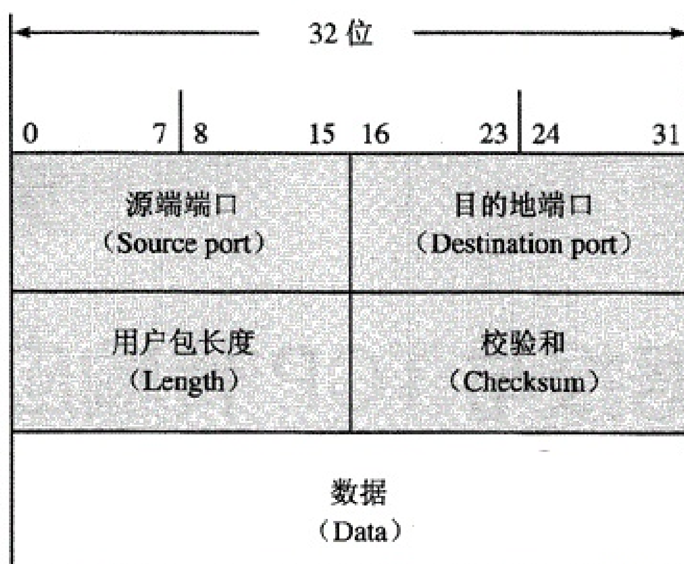
TCP和UDP还有另外一个区别。(2)UDP不能保证有序传输。即UDP不能确保数据的发送和接收顺序。

下面就来看看UDP协议的工作原理，对UDP的工作原理有一个好的理解，对后面介绍的UDP编程也是一个好的基础。

1.1 UDP的工作原理

UDP将网络数据流量压缩成数据报的形式，每一个数据报用8个字节（8 X 8位=64位）描述报头信息，剩余字节包含具体的传输数据。UDP报头（只有8个字节）相当于TCP的报头（至少20个字节）很短，UDP报头由4个域组成，每个域各占2个字节，具体为源端口、目的端口、用户数据报长度和校验和，

具体结构见下图（下面也贴出了TCP报文的结构图，与UDP数据报做一个对比的作用）：





UDP协议和TCP协议都使用端口号为不同的应用保留其各自的数据传输通道这一机制，数据发送方将UDP数据报通过源端口发送出去，而数据接收方则通过目标端口接收数据。

1.2 UDP的优势

前面介绍中说UDP相对于TCP是不可靠的，不能保证有序传输的传输协议，然而UDP协议相对于TCP协议的优势在哪里呢？，

UDP相对于TCP的优势主要有三个方面的：

（1）UDP速度比TCP快。

由于UDP不需要先与对方建立连接，也不需要传输确认，因此其数据的传输速度比TCP快很多。对于一些着重传输性能而不是传输完整性的应用（网络音频播放、视频点播和网络会议等），使用UDP协议更加适合，因为它传输速度快，使通过网络播放的视频音质好、画面清晰。

（2）UDP有消息边界。

通过UDP协议进行传输的发送方对应用程序交下来的报文，在添加首部后就向下直接交付给IP层。既不拆分也不合并，而是保留这些报文的边界，所以使用UDP协议不需要像TCP那样考虑消息边界的问题，这样就使得UDP编程相对于TCP在接收到的数据处理方面要简单的多。（对于TCP消息边界的问题可以查看相关的文档，在这里我就不列出来了）

（3）UDP可以一对多传输

由于传输数据部建立连接，也就不需要维护连接状态，因此一台服务器可以同时向多个客户端发送相同的信息。利用UDP可以使用广播或者组播的方式同时向子网的所有客户端进程发送信息，广播和组播的介绍放到后面TCP编程中介绍。

上面介绍了UDP协议相对于TCP协议的优势，其中速度快是UDP的最重要的优势，也是像一些网络会议、即时通信软件传输层选择UDP协议进行传输的原因所在。

二、.net平台对UDP编程的支持

介绍完UDP相对于TCP的优势后，当然很希望在.net平台下开发一个基于UDP协议的一个应用了，然后.net平台下对UDP编程也做了很好的支持，为我们开发基于UDP协议的网络应用提供很多方便之处，下面就简单介绍.net平台下对UDP编程的支持（主要介绍提供的类来对UDP协议进行编程）。

.net类库中的UdpClient类对基础的Socket进行了封装，这样就在发送和接受数据时不需要考虑底层套接字的收发时处理的一些细节问题，这样为UDP编程提供了方便，也可以提高开发效率（感觉net就是做这样的事情的，对一些底层的实现进行封装，方便我们的调用，这也体现了面向对象语言的封装特性）对于这个的具体的使用我就不做过多的介绍的，在后面的UDP编程的实现部分将会对该类中主要方法的使用，大家可以查看MSDN来查看该类中其他成员的使用：
http://msdn.microsoft.com/zh-cn/library/System.Net.Sockets.UdpClient.aspx

三、UDP编程的具体实现

由于UDP进程在通信之前是不需要建立连接，消息接收方可能并不知道是谁给它发的消息，因此UDP编程分为两种模式：一种“实名发送”，即接收方可以由收到的消息得知发送方进程端口，另外一种则为“匿名发送”，即接收方并不知道发给它信息的远程进程究竟来自哪个端口。下面通过一个winform 程序来演示下UDP的编程：

实现代码：

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Threading;
using System.Windows.Forms;

namespace UDPClient
{
    public partial class frmUdp : Form
    {
```

```
        private UdpClient sendUdpClient;
        private UdpClient receiveUpdClient;
        public frmUdp()
        {
            InitializeComponent();
            IPAddress[] ips = Dns.GetHostAddresses("");
            tbxlocalip.Text = ips[3].ToString();
            int port = 51883;
            tbxlocalPort.Text = port.ToString();
            tbxSendtoIp.Text = ips[3].ToString();
            tbxSendtoport.Text = port.ToString();
        }

        // 接受消息
        private void btnReceive_Click(object sender, EventArgs e)
        {
            // 创建接收套接字
            IPAddress localIp = IPAddress.Parse(tbxlocalip.Text);
            IPEndPoint localIpEndPoint = new IPEndPoint(localIp, in
            receiveUpdClient = new UdpClient(localIpEndPoint);

            Thread receiveThread = new Thread(ReceiveMessage);
            receiveThread.Start();
        }

        // 接收消息方法
        private void ReceiveMessage()
        {
            IPEndPoint remoteIpEndPoint = new IPEndPoint(IPAddress.
            while (true)
            {
                try
                {
                    // 关闭receiveUdpClient时此时会产生异常
                    byte[] receiveBytes = receiveUpdClient.Receive(

                    string message = Encoding.Unicode.GetString(rec

                    // 显示消息内容
                    ShowMessageforView(lstbxMessageView, string.For
                }
                catch
                {
                    break;
                }
            }
        }

        // 利用委托回调机制实现界面上消息内容显示
        delegate void ShowMessageforViewCallBack(ListBox listbox, s
        private void ShowMessageforView(ListBox listbox, string tex
        {
            if (listbox.InvokeRequired)
```

```
            {
                ShowMessageforViewCallBack showMessageforViewCallba
                listbox.Invoke(showMessageforViewCallback, new obje
            }
            else
            {
                lstbxMessageView.Items.Add(text);
                lstbxMessageView.SelectedIndex = lstbxMessageView.I
                lstbxMessageView.ClearSelected();
            }
        }
        private void btnSend_Click(object sender, EventArgs e)
        {
            if (tbxMessageSend.Text == string.Empty)
            {
                MessageBox.Show("发送内容不能为空","提示");
                return;
            }

            // 选择发送模式
            if (chkbxAnonymous.Checked == true)
            {
                // 匿名模式(套接字绑定的端口由系统随机分配)
                sendUdpClient = new UdpClient(0);
            }
            else
            {
                // 实名模式(套接字绑定到本地指定的端口)
                IPAddress localIp = IPAddress.Parse(tbxlocalip.Text
                IPEndPoint localIpEndPoint = new IPEndPoint(localIp
                sendUdpClient = new UdpClient(localIpEndPoint);
            }

            Thread sendThread = new Thread(SendMessage);
            sendThread.Start(tbxMessageSend.Text);
        }

        // 发送消息方法
        private void SendMessage(object obj)
        {
            string message = (string)obj;
            byte[] sendbytes = Encoding.Unicode.GetBytes(message);
            IPAddress remoteIp = IPAddress.Parse(tbxSendtoIp.Text);
            IPEndPoint remoteIpEndPoint = new IPEndPoint(remoteIp,
            sendUdpClient.Send(sendbytes, sendbytes.Length, remoteI

            sendUdpClient.Close();

            // 清空发送消息框
            ResetMessageText(tbxMessageSend);
        }

        // 采用了回调机制
```

```
        // 使用委托实现跨线程界面的操作方式
        delegate void ResetMessageCallback(TextBox textbox);
        private void ResetMessageText(TextBox textbox)
        {
            // Control.InvokeRequired属性代表
            // 如果控件的处理与调用线程在不同线程上创建的，则为true,否则为fa
            if (textbox.InvokeRequired)
            {
                ResetMessageCallback resetMessagecallback = ResetMe
                textbox.Invoke(resetMessagecallback, new object[] 
            }
            else
            {
                textbox.Clear();
                textbox.Focus();
            }
        }

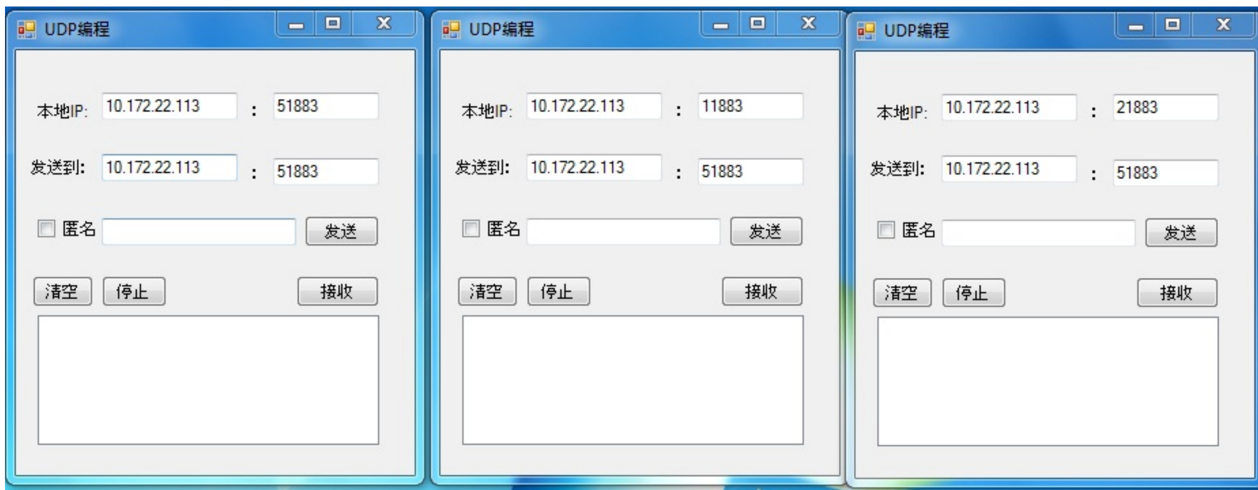
        // 停止接收
        private void btnStop_Click(object sender, EventArgs e)
        {
            receiveUpdClient.Close();
        }

        // 清空接受消息框
        private void btnClear_Click(object sender, EventArgs e)
        {
            this.lstbxMessageView.Items.Clear();
        }
    }
}
```

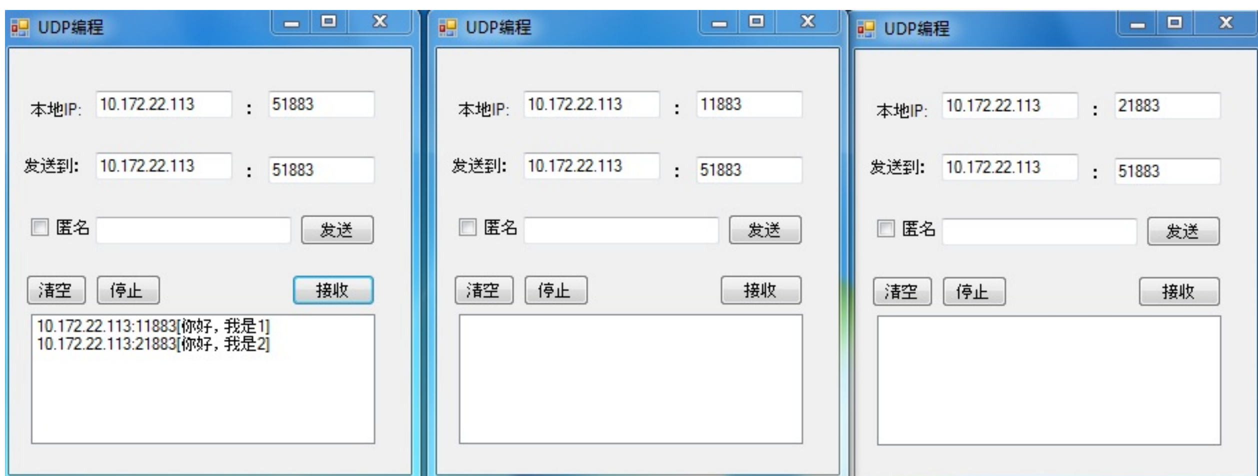
运行结果：

实名发送：

在本地运行本程序的三个进程（分别为A,B,C）,把进程C做为接受进程，进程A和进程B都向进程C发信息，进程A和进程分别绑定端口号为11883和21883，发送到端口都为51883，配置界面如下：



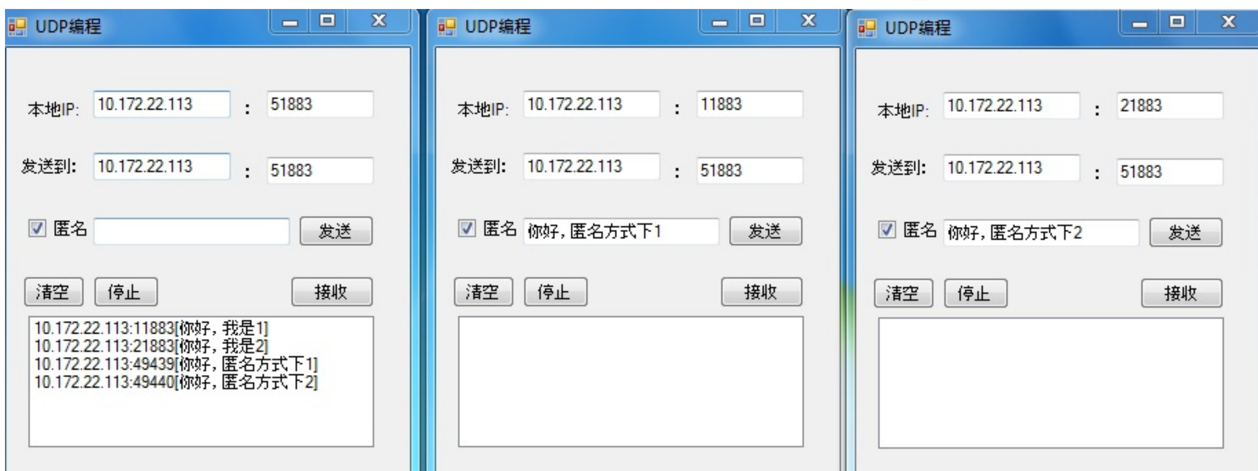
首先不勾选“匿名”复选框，在进程C中点击“接收”按钮开启接受线程，在A进程和B进程中发送消息框里分别输入你好，我是**1**和你好，我是**2** ,然后点击发送按钮，此时在进程中就可以看到进程A和进程B发来的消息，如下图：



从图中可以看出每条消息之前都显示了消息的准确来源（包括消息进程锁在的Ip地址和端口号）

匿名发送：

下面把“匿名”复选框勾上后，再按照前面的步骤将得到下面的结果：



从图中结果可以看出此时列表中显示的消息来源的进程端口号分别为49439和49440，而不是发送消息进程的真实端口（11883和21883）

这种UDP只能辨别消息源主机的Ip地址，而无法知道发消息的进程究竟是哪个端口称为“匿名发送”。正如我们平时发手机短信一样，如果我们把认识的名字和电话号码预先存在通讯录里，当一发来信息，接受方马上就可以从来电显示中看到是谁发来的（实名模式）;但是如果是陌生人发来信息或者广告等信息时，仅看来电显示，根本不知道对方是谁（匿名模式），QQ发消息也是一样的道理。

四、**UDP**广播和组播

前面UDP的实现中发送数据使用的都是一对一（单播）的通信方式，即只将数据发送到某一个进程。前面提到UDP可以实现一对多的传输方式，即通过广播和组播把数据发送给一组进程。下面就介绍下UDP广播和组播的相关知识。

4.1 广播和组播的基本概念

虽然利用TCP协议可以保证数据的可靠、有序的传输，但是TCP仅支持一对以的传输，而且传输时需要在发送端和每一个接受端之间建立单独的数据通信通道，如果需要实现网络会议、网络视频的点播等功能时要向大量主机发送相同的数据包，如果采用单播方式逐个节点传输的话，将会给发送方带来网络堵塞等问题，此时可以考虑实现UDP的多播方式——即广播和组播来实现这样的功能（一对多通信分为广播和组播两种形式）。

广播是指同时向子网中的多台计算机发送消息，并且所有子网中的计算机都可以接收到发送方发来的消息，每个广播消息包含一个特殊的IP地址，这个IP的中子网内主机标志部分的二进制都为**1**，例如，子网掩码为255.255.255.0，对于子网192.168.0，则这个IP地址为192.168.0.255.

然后广播消息又分为本地广播和全球广播两种类型， 本地广播是指向子网中的所有计算机发送广播消息，其他网络不会受到本地广播的影响。

IP地址分为两部分——网络标志部分和主机标志部分，这两部分是靠子网掩码来区分的，主机标记部分二进制全部为1的地址成为本地广播地址。例如：

A类网络192.168.0.0，使用子网掩码255.255.0.0，则本地广播地址为：



对于IPv4来说，全球广播使用所有位全为1的IP地址，即255.255.255.255，这个广播地址代表数据报的目的地是网络上所有设备，但是由于路由器会自动过滤全球广播，所以使用这个地址根本就没有任何意义。

然后当接收者分布于多个不同的子网时，广播将不再适用，此时可以通过组播的方式来实现，组播也叫多路广播，组播是将信息从一台计算机发送到本网或全网内指定的计算机上，即发送到那些加入了指定组播组的计算机上，每台计算机都可以通过程序随时加入某个组播组中，也可以随时退出来， 就像我们开网了会议一样，可以随时加入会议室进行开会，会议结束和会议进行中都可以随意的退出来。

4.2 加入和退出组播组

组播组又称为多路广播组，组播地址的范围在224.0.0.0到239.255.255.255的D类IP地址（至于这个概念大家可以百度百科里面就查看）。任何发送到组播地址的消息都会被发送到组内所有成员设备上，组可以使永久的也可以是临时，大多数我们使用的都是临时的，仅在有成员的时候才存在。

使用组播时，注意生命周期（TTL，Time to live）的设，TTL值表示允许路由器转发的最大次数，当达到这个最大值时，数据包就会被丢弃，TTL的默认值为1，设置为1时表明只能在子网中发送数据

加入组播组：

UdpClient类提供了**JoinMulticastGroup**方法，用于将UdpClient加入到使用指定的IPAddress的组播组中，调用该方法后，基础的Socket会自动向路由器发送数据包，用于请求成为组播组的成员，如果成为组播组成员，就可以接收该组播组的数据报。至于具体方法的时候会在后面实现UDP广播程序中会用到，另外大家也可以查看MSDN，所以这里我就不再列出来了，只是指出这个方法的作用，让大家知道有这么个方法来调用。

退出组播组：

同样利用UdpClient的DropMulticastGroup方法，可以退出组播组，调用该方法后，基础Socket会自动向路由器发送数据包，用于请求从指定的组播组里退出，从组中回收UdpClient对象之后，将不再接受发送到该组播组的数据报。

五、总结

由于时间的关系，这篇文章就介绍到这里的，至于实现UDP广播的程序放在后面一个专题里面的，前面也对广播和组播的概念进行了简单的介绍，相信大家也对广播和组播有了个简单的认识（广播组和组播组说白了就是一个IP地址的集合，其实实现UDP广播的程序和前面实现单播的程序差不多，只是前面绑定了一个IP地址当然也只能发送到一个IP地址了，也就是所谓的单播，多播和广播就是发送的IP地址是一个组，当然也就实现了一对多的传输了）。UDP广播程序的实现就放在下一个专题和大家分享的，因为我现在要去吃饭了，吃完饭再继续和大家介绍，希望大家如果觉得有帮助的话，也可以推荐下，这给我继续写下去的动力，谢谢大家的支持

本专题源码地址:http://files.cnblogs.com/zhili/UDPCommunication.zip

[C# 网络编程系列]专题七：UDP编程补充——UDP广播程序的实现

上次因为时间的关系，所以把上一个专题遗留下的一个问题在本专题中和大家分享下，本专题主要介绍下如何实现UDP广播的程序，下面就直接介绍实现过程和代码以及运行的结果。

一、程序实现

UDP广播程序的实现代码：

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Threading;
using System.Windows.Forms;

namespace UDPBroadcast
{
    /// <summary>
    /// 在界面上，用户可以设置本地进程的IP地址和端口号，并将地址加入某个组播组
    /// 可以输入发送消息的目的组的地址，并且勾选“广播”复选框将采用广播的方式发
    /// 在界面上点击“接受按钮”就启动接收线程，这样程序就可以接收广播或组播的信
    /// </summary>
    public partial class UdpBroadcasefrm : Form
    {
        private UdpClient sendUdpClient;
        private UdpClient receiveUdpClient;
        // 组播IP地址
        IPEndPoint broadcastIpEndPoint;
        public UdpBroadcasefrm()
        {
            InitializeComponent();
            IPAddress[] ips = Dns.GetHostAddresses(Dns.GetHostName(
            tbxlocalip.Text = ips[5].ToString();
            tbxlocalport.Text = "8002";
            // 默认组,组播地址是有范围
            // 具体关于组播和广播的介绍参照我上一篇博客UDP编程
            // 本地组播组
            tbxGroupIp.Text = "224.0.0.1";
            // 发送到的组播组
            tbxSendToGroupIp.Text = "224.0.0.1";
        }

        // 设置加入组
        private void chkbxJoinGtoup_Click(object sender, EventArgs
        {
            if (chkbxJoinGtoup.Checked == true)
```

```
            {
                tbxGroupIp.Enabled = false;
            }
            else
            {
                tbxGroupIp.Enabled = true;
                tbxGroupIp.Focus();
            }
        }

        // 选择发送模式后设置
        private void chkbxBroadcast_Click(object sender, EventArgs
        {
            if (chkbxBroadcast.Checked == true)
            {
                tbxSendToGroupIp.Enabled = false;
            }
            else
            {
                tbxSendToGroupIp.Enabled = true;
                tbxSendToGroupIp.Focus();
            }
        }

        // 发送消息
        private void btnSend_Click(object sender, EventArgs e)
        {
            if (tbxMessageSend.Text == "")
            {
                MessageBox.Show("消息内容不能为空！","提示");
                return;
            }

            // 根据选择的模式发送信息
            if (chkbxBroadcast.Checked == true)
            {
                // 广播模式(自动获得子网中的IP广播地址)
                broadcastIpEndPoint = new IPEndPoint(IPAddress.Broa
            }
            else
            {
                // 组播模式
                broadcastIpEndPoint = new IPEndPoint(IPAddress.Pars
            }

            // 启动发送线程发送消息
            Thread sendThread = new Thread(SendMessage);
            sendThread.Start(tbxMessageSend.Text);
        }

        // 发送消息
        private void SendMessage(object obj)
        {
```

```
            string message = obj.ToString();
            byte[] messagebytes = Encoding.Unicode.GetBytes(message
            sendUdpClient = new UdpClient();
            // 发送消息到组播或广播地址
            sendUdpClient.Send(messagebytes, messagebytes.Length, b
            sendUdpClient.Close();

            // 清空编辑消息框
            ResetMessageText(tbxMessageSend);
        }

        // 利用委托回调机制来实现界面上的消息清空操作
        delegate void ResetMessageTextCallBack(TextBox textbox);
        private void ResetMessageText(TextBox textbox)
        {
            if (textbox.InvokeRequired)
            {
                ResetMessageTextCallBack resetMessageCallback = Res
                textbox.Invoke(resetMessageCallback, new object[] {
            }
            else
            {
                textbox.Clear();
                textbox.Focus();
            }
        }

        // 接收消息
        private void btnReceive_Click(object sender, EventArgs e)
        {
            chkbxJoinGtoup.Enabled = false;
            // 创建接收套接字
            IPAddress localIp = IPAddress.Parse(tbxlocalip.Text);
            IPEndPoint localIpEndPoint = new IPEndPoint(localIp, in
            receiveUdpClient = new UdpClient(localIpEndPoint);
            // 加入组播组
            if (chkbxJoinGtoup.Checked == true)
            {
                receiveUdpClient.JoinMulticastGroup(IPAddress.Parse
                receiveUdpClient.Ttl = 50;
            }

            // 启动接受线程
            Thread threadReceive = new Thread(ReceiveMessage);
            threadReceive.Start();
        }

        // 接受消息方法
        private void ReceiveMessage()
        {
            IPEndPoint remoteIpEndPoint = new IPEndPoint(IPAddress.
            while (true)
            {
```

```
                try
                {
                    // 关闭receiveUdpClient时此时会产生异常
                    byte[] receiveBytes = receiveUdpClient.Receive(
                    string receivemessage = Encoding.Unicode.GetStr

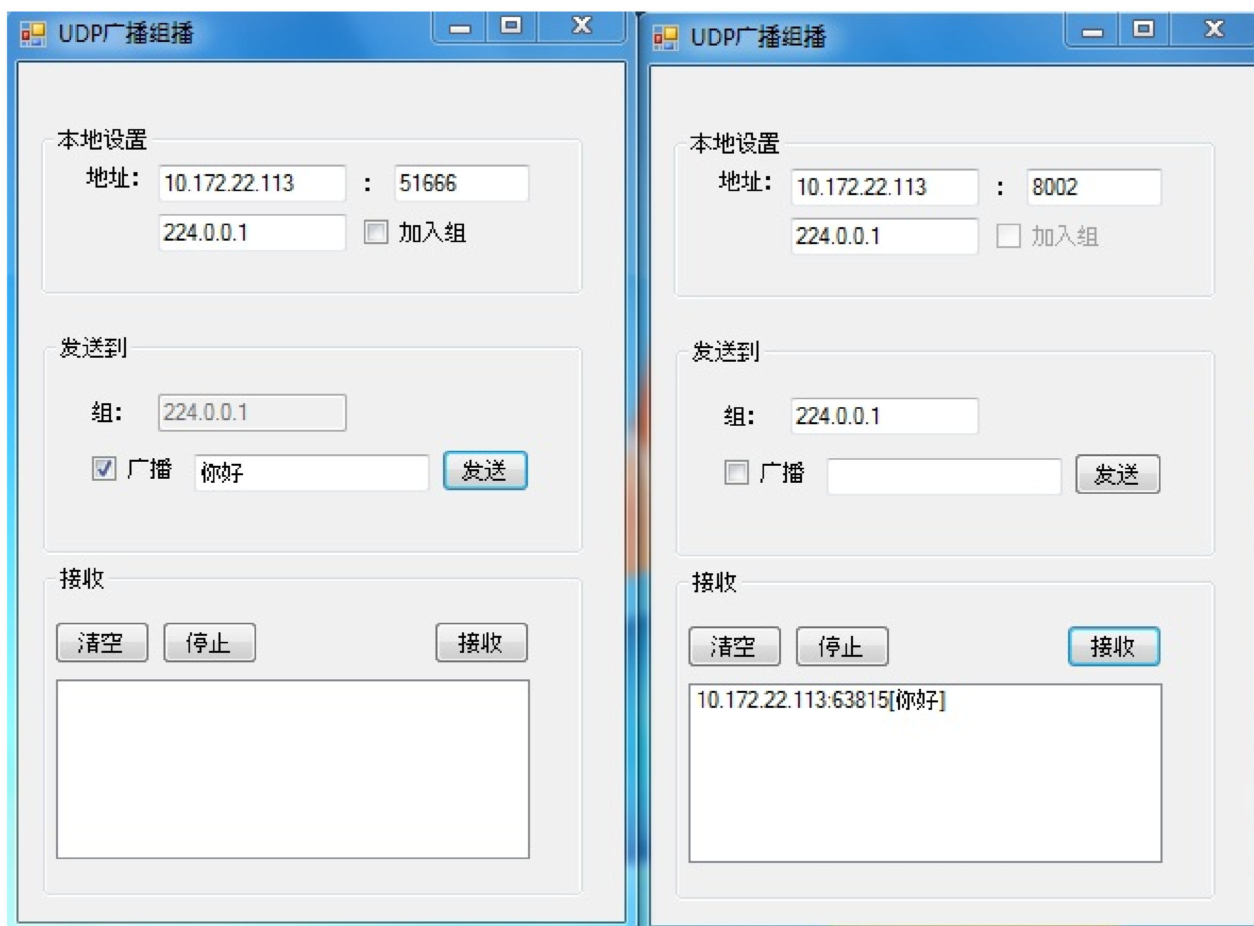
                    // 显示消息内容
                    ShowMessage(lstMessageBox, string.Format("{0}[
                }
                catch
                {
                    break;
                }
            }
        }
        // 通过委托回调机制显示消息内容
        delegate void ShowMessageCallBack(ListBox listbox,string te
        private void ShowMessage(ListBox listbox, string text)
        {
            if (listbox.InvokeRequired)
            {
                ShowMessageCallBack showmessageCallback = ShowMessa
                listbox.Invoke(showmessageCallback, new object[] {
            }
            else
            {
                listbox.Items.Add(text);
                listbox.SelectedIndex = listbox.Items.Count - 1;
                listbox.ClearSelected();
            }
        }

        // 清空消息列表
        private void btnClear_Click(object sender, EventArgs e)
        {
            lstMessageBox.Items.Clear();
        }

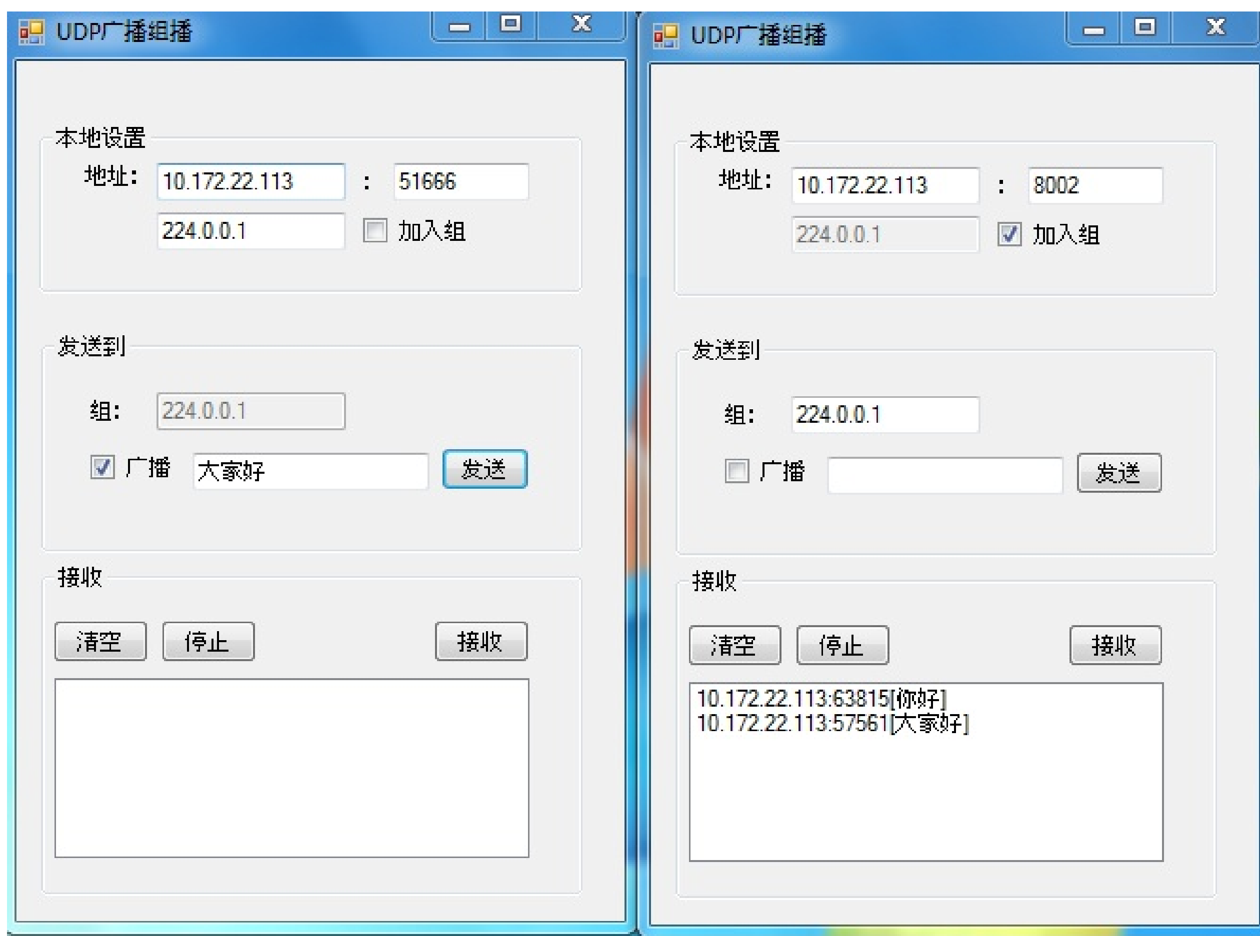
        // 停止接收
        private void btnStop_Click(object sender, EventArgs e)
        {
            chkbxJoinGtoup.Enabled =true;
            receiveUdpClient.Close();
        }

    }
}
```

广播演示结果（接收端直接点接收按钮后开启接受线程，在发送端勾选“广播选项”输入发送信息点发送按钮后的界面如下）：

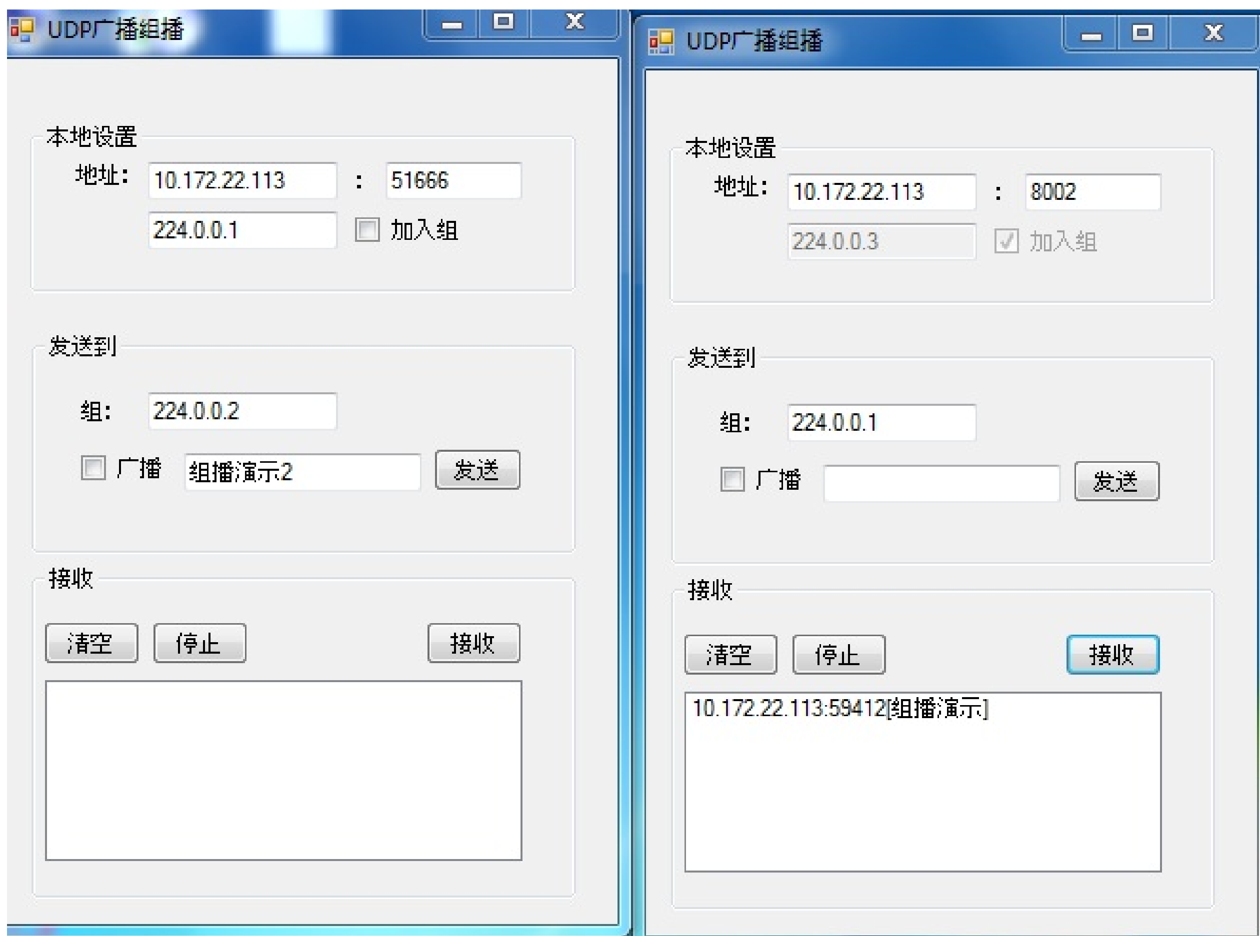


下面通过把接收端加入组后的结果，首先终止接收线程，然后勾选“加入组”复选框，然后单击“接收”按钮重新开启接收线程，输出结果如下：



从广播演示的两个情况可以看出广播消息会同时向网上的一切进程转发，无论这个进程是独立的还是加入了某个组播组中的进程，都可以接收广播消息

下面演示下组播的结果：



如果把接收端的组地址改为224.0.0.3时，此时发送端发送的消息"组播演示2"将不会发送到不同的组播地址，则接收端就接收不到此时的消息。

从组播结果中可以看出只有加入组播地址224.0.0.2的进程才能接收到信息。

需要注意的地方是：从前面的截图中可以看出，不论是广播还是组播，仅仅从收到的信息无从知道发送给它的进程的端口号，所以广播和组播消息都是匿名发送，并且通过对UDP广播和组播的理解可以简单实现一个消息群发的功能（QQ的群里聊天就是这个原理）。

二、 总结

本专题主要是针对上一专题的补充——实现一个简单的UDP广播（组播）程序，通过这样一个发送端可以发送给在组播地址中的所有用户和所有子网中的所有用户。本专题可以说是对UDP编程的一个扩充吧，希望大家看了本专题后可以对ＵＤＰ协议有大致的理解。在下一个专题中会和大家介绍下P2P编程的相关知识。

全部源码地址：http://files.cnblogs.com/zhili/UDPBroadcast.zip

[C# 网络编程系列]专题八：P2P编程

引言：

前面的介绍专题中有朋友向我留言说介绍下关于P2P相关的内容的，首先本人对于C#网络编程也不是什么大牛，因为能力的关系，也只能把自己的一些学习过程和自己的一些学习过程中的理解和大家分享下的，下面就进入正题——P2P（Peer to Peer）编程

一、**P2P**的介绍

首先，现在大家熟知的BT、电驴、迅雷、QQ、MSN和PPlive等都是基于P2P方式实现的软件，并且对等联网（Peer to Peer，P2P）将是互联网的发展方向，因此对于P2P技术的了解显得非常的重要，下面就来介绍下P2P架构：

在P2P技术之前，我们所有的网络应用都采用C/S或者B/S架构来实现的，然而在之前C/S架构的应用程序中，客户端软件向服务器发出请求，服务器然后对客户端请求做出响应，在这种情况下，如果客户端越多，此时服务器的压力就越大。然而采用P2P技术实现的每台计算机既是客户端，也是服务器，他们的功能都是对等的。对于安装了P2P软件（如迅雷，QQ等）的计算机加入一个共同的P2P网络，网络中的节点之间可以直接进行数据传输和通信。

1.1 P2P架构和**C/S**架构的比较

C/S架构有下面的缺点（其实上面的简单介绍中也讲到过）：

1. 服务器负担过重。当大量用户访问C/S系统的服务器时，服务器常常会出现网络堵塞等现象，这时候，我们可能会通过增加投资提高服务器的硬件性能

2. 系统稳健性和服务器关联密切。指的是——如果服务器出现了问题时，整个系统的运行将会瘫痪（感觉是面向对象中经常强调的原则——低耦合原则）

然而P2P具有下面的特点：

1.对等模式

P2P系统中的客户端能够同时扮演客户端和服务器的角色，使两台计算机之间能够不通过服务器直接进行信息分享（QQ中当好友在线的时候发信息时，相信此时是不需要经过服务器转发的，只有当给离线好友发送消息时，此时应该会先把消息发送到服务器端存储起来，当好友再次登录的时候，会和服务器进行连接，服务器会进行判断是不是给这个用户的信息来决定是否转发，QQ软件的实现属于混合型P2P结构的， 这个会在后面的P2P系统分类中介绍。）

注：括号中都是我个人的一些理解，如果有什么说错的地方请大家及时更正我，这样我会及时的更新以免误导大家，谢谢大家监督。

2. 网络资源的分布式存储

在C/S架构中，所有客户端都直接从服务器下载所有数据资源，这样势必会加重服务器的负担，而P2P则改变了以服务器为中心的状态，使每个节点可以先从服务器上个下载一部分，然后再相互从对方或者其他节点下载其余部分。采用这种方式，当大量客户端同时下载时，就不会形成网络堵塞现象了。

1.2 P2P系统的分类

使用P2P技术的系统分为两类：（1）单纯型P2P——没有专用的服务器。安装了P2P软件的各个计算机可以直接通信

（2）混合型P2P——有专用的服务器，此时的服务器一般叫索引服务器，此服务器与C/S架构下的服务器不同，在C/S架构下所有资源都存储在服务器中，所有传递的信息都要经过服务器，而在混合型P2P系统中的索引服务器仅仅起到协调和扩展的功能，资源不是全部存储在服务器上，而是分布在各个电脑上，安装了P2P软件的电脑开始全部和索引服务器连接，以便告知自己监听的IP地址和端口号，然后再通过索引服务器告诉其他与自己连接的电脑，每台计算机的连接和断开都通过服务器通知网络上有联系的计算机，这样就减轻了每台计算机搜索其他计算机的负担，但是信息的传递还是通过点对点的方式来完成（这里可以以QQ为例，当我们电脑上安装了QQ这类P2P软件时，安装了QQ这类软件的计算机就会加入一个P2P网络，并且登陆的时候都会与索引服务器建立连接，通过连接来告诉服务器自己的IP地址和端口号，当我们找一个好友聊天时，此时自己的计算机和好友的计算机都会与服务器端口连接，但是要互相发送消息，自己的计算机必须知道好友计算机的IP地址和端口号才可以通信，这样的工作正是通过索引服务器来告知对方的--指的是告诉自己的计算机好友的计算机的IP地址和端口号，告诉好友的计算机自己的IP地址和端口号，这样双方就可以不通过服务器直接通信了。）

1.3 主流**P2P**应用分类

P2P 网络应用大致可以分为三类—— 1. 文件共享类，例如迅雷，BT等软件都是文件共享类的应用

2. 即时通信类，例如QQ，MSN等软件都是属于即时通信类

3. 多媒体传输类，例如在线视频直播软件，PPlive等软件

从上面的分类可以看出，现在网络上流行的软件都采用了P2P技术来实现的， 但是它们的实现肯定不是单纯的只采用P2P技术来实现的， 而是采用多种技术来实现的， 在下一专题中将介绍利用TCP，UDP和P2P等技术来实现类似QQ的一个即时通信程序，希望通过此程序可以综合前面专题介绍的内容以及帮助大家对QQ等软件的实现原理有了解。

二、**P2P**的基本原理

在前面我们对P2P的一些知识进行的简单的介绍， 通过前面的介绍相信大家对P2P的技术有了一定的了解，但是要自己开发一个P2P的应用当然必须了解P2P技术的实现原理的，下面就介绍下P2P实施的基本原理。

安装了P2P软件后，首先双方要进行通信，必须能够发现对方（指的就是知道对方的IP地址和端口号），一旦发现了对方后才可以进行通信，所以P2P应用程序一般分为发现、连接和通信3个阶段。 发现阶段负责动态定位通信方的网络位置；连接阶段负责在双方建立网络连接，通信阶段负责在双方之间传输数据。

2.1 发现阶段

一台计算机要和另外一台计算机通信，必须知道对方的IP地址和监听端口，否则就无法向对方发送消息。在之前的C/S架构中，服务器的IP地址一般固定不变的，并且提供服务的计算机域名也一般不会改变，所以为了方便客户端访问，一些Web服务器在DNS（DNS其实就是域名和IP地址的一个映射）中进行了注册，客户机可以利用域名解析机制将服务器域名解析为IP地址，然后在P2P应用中，各个对等节点（计算机或资源）可以随时加入和随时离开，并且对等节点的IP地址也不是固定的，所以不能采用DNS的机制来获取P2P架构中的对等节点的信息。

目前，在单纯型P2P中，针对如何发现对等节点，各种P2P技术采用的协议和标准都不一样，微软在.net 支持对等名称解析协议（Peer name Resolution Protocol, PNRP）,该协议可以发现对等节点的信息，通过无服务器的解析功能将任何资源解析为一组ＩＰ地址和端口号，在后面的实现的简单程序用的就是这个协议来完成发现阶段的。

2.2 连接和通信阶段

完成对等节点的发现后，接下来就可以根据需要，选择TCP、UDP或者其他协议完成数据传输。如果选择TCP，则需要先建立连接，再利用该连接传输数据，关于TCP的内容可以查看我之前的专题；如果选择UDP，则无须建立连接，直接在对等节点之间通信就可以了。

三、.net平台对P2P编程的支持

之前在发现阶段也介绍了.Net平台对P2P编程的支持的，然后微软帮我们已经封装好了对PNRP协议的实现，这些类在System.Net.PeerToPeer命名空间里（微软现在很多东西都帮我们封装了，这样可以方便我们开发应用程序，感觉微软的做的东西都是这样，把程序的实现都帮我们做好了，我们开发程序的时候只要关注业务逻辑就好了的， 这样做当然有好处也有坏处的， 我觉得好处就是缩短软件的开发周期，让花更多的时间去实现软件真真的业务逻辑方面的东西，不好的地方就是现在的程序员就不能叫程序员，所以园子里面有很多人都称码农,这些我自己的观点了）

3.1 对等名称解析协议（PNRP）

PNRP可以完成对等名称的注册和解析（可以和DNS对比来理解）。

3.1.1 基本概念

第一个介绍的是对等名称的概念，我们将每一个网络资源（包括计算机，P2P应用程序、视频、Mp3或其他文档等资源）抽象为对等节点，对等节点名称当然就是对等节点的名称。对等节点名称简称为对等名，分为安全的和不安全的两种形式，不安全的名称仅由文本字符串组成，任何人都可以注册一个相同的不安全对等名称；安全的由一个公钥/私钥（代表唯一）对支持，所以使用PNRP注册时，不会受到欺骗，对等名称的格式如下：

Authority.Classifier

Authority的值取决于该名称的安全类型。对于不安全的类型，Authority为单字符“0”，而对于安全的对等名称，Authority由40个十六机制字符组成

Classifier是用户定义的用于标志对等节点的字符串，最大长度为150个Unicode字符。例如，对等名称0.PeerNametest1就是一个不安全的对等名称。

第二个概念就是云（Cloud）的概念，安装了相同P2P软件的计算机会加入一个共同的P2P网络中，才能相互识别各自拥有的资源并顺利进行P2P通信。微软PNPR协议将这个P2P网络称为“云”。云是指一组可以通过**P2P**网络相互识别的对等节点及其上资源的集合。云中的所有对等节点都可以解析注册到该云中的其他任何资源所在的位置（IP+Port),一个对等节点上的某个资源可以同时注册到多个云中。

PNPR目前使用了两种云——本地云和全局云。一个对等名称若注册到链接一本地云，就意味着只有同一本地网络上的其他对等节点可以解析该名称。而注册到全局云上的对等名称则允许IPv6互联网的任何对等节点解析。

注：全局云是基于IPv6协议的，并不支持IPv4，如果不存在IPv6地址，则不会出现全局云，也无法加入全局云。由于现在网上绝大多数应用使用的仍然是IPv4的地址，所以我们通常的P2P编程还用不到全局云，而只能使用默认的本地云。

3.1.2 名称注册

任何资源要被网络上的其他计算机识别到，首先必须注册进P2P网络，名称注册就是将包含对等节点信息的对等名称发布到云中，以便其他对等节点解析。一个资源如果注册到云中后，就可以被云中的其他对等节点解析和访问。

关于名称注册的具体内容，在后面的P2P的程序中也会使用的， 相信大家可以通过代码来进一步理解名称的注册，这里就先介绍到这里的

3.1.3 名称解析

名称解析是指利用对等名称获取到云中资源所在对等节点的IP地址和端口号的过程（和DNS解析原理一样）。**PNPR**名称解析仅能够注册到云中的其他对等节点资源，而不能发现自身注册的资源

PNPR协议没有使用索引服务器，所以为了完成解析，云中的每个对等节点都存储一些PNRP ID的缓存记录。PNPR缓存中的都含有PNPR ID和应用程序的IP地址和端口号。名称解析的步骤为——首先在本地计算机对等节点的缓存中查找目标资源，如果没有，则在缓存中的临近节点查看，这样循环下去，直到找到目标资源所在的对等节点位置。

3.2 PeerToPeer命名空间

上面主要介绍了PNPR协议的工作过程（相当于是理论部分了），下面就介绍下.net 为我们封装好PNPR的类的使用。这里就简单指明几个常用类的使用，并附上MSDN的链接，大家可以直接点链接进行查看详细内容，因为后面的P2P程序中也有具体的使用，所以这里就不一一列出来了。

类名	MSDN链接
Peer类	http://msdn.microsoft.com/zh-cn/library/system.net.pee
Cloud类	http://msdn.microsoft.com/zh-cn/library/system.net.pee
PeerName类	http://msdn.microsoft.com/zh-cn/library/system.net.pee
PeerNameRecord类	http://msdn.microsoft.com/zh-cn/library/system.net.pee
PeerNameRegistration类	http://msdn.microsoft.com/zh-cn/library/system.net.peertopeer.peernameregistration
PeerNameResolver类	http://msdn.microsoft.com/zh-cn/library/system.net.pee

这些类基本上从类名都可以大致知道他们的用途的，所以在这里就没有一一介绍的，只是附上了MSDN的链接。

四、实现**P2P**应用程序

以上介绍了那么多P2P的相关的知识，主要是为了实现一个自定义的P2P应用程序做准备的，这里就简单实现了资源发现的一个程序。

核心代码：

对等名称的注册代码：

```
// 注册资源
        private void btnRegister_Click(object sender, EventArgs e)
        {
            if (tbxResourceName.Text == "")
            {
                MessageBox.Show("请输入发布的资源名！", "提示");
                return;
            }

            // 将资源名注册到云中
            // 具体资源名的结构在博客有介绍
            PeerName resourceName = new PeerName(tbxResourceName.Te
            // 用指定的名称和端口号初始化PeerNameRegistration类的实例
            resourceNameReg[seedCount] = new PeerNameRegistration(r
            // 设置在云中注册的对等名对象的其他信息的注释
            resourceNameReg[seedCount].Comment =resourceName.ToStri
            // 设置PeerNameRegistration对象的应用程序定义的二进制数据
            resourceNameReg[seedCount].Data = Encoding.UTF8.GetByte
            // 在云中注册PeerName(对等名)
            resourceNameReg[seedCount].Start();
            seedCount++;
            comboxSharelist.Items.Add(resourceName.ToString());
            tbxResourceName.Text = "";
        }
```

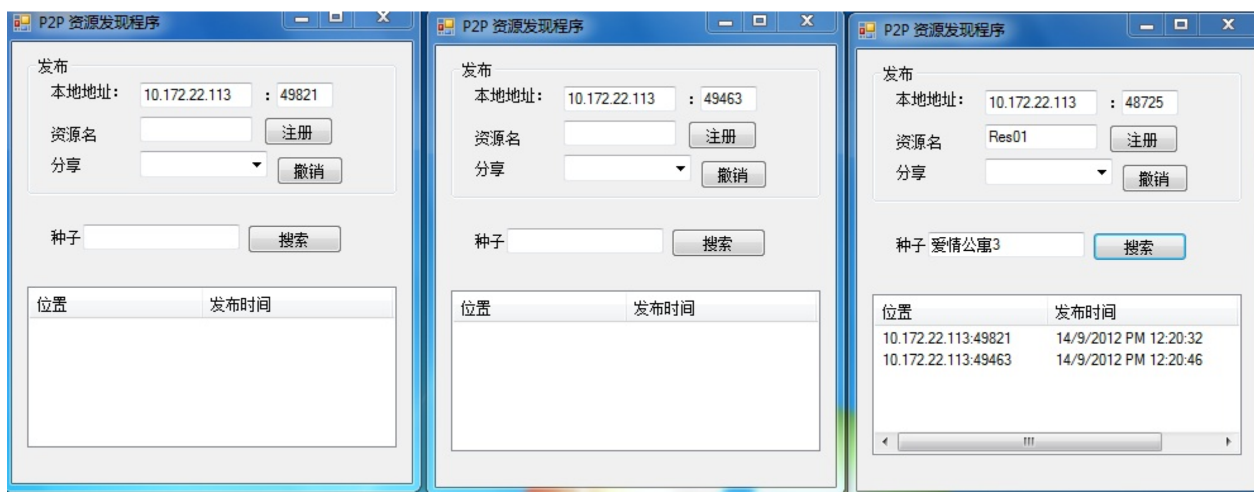
名称解析代码（搜索资源）：

```
// 搜索资源
        private void btnSearch_Click(object sender, EventArgs e)
        {
            if (tbxSeed.Text == "")
            {
                MessageBox.Show("请先输入要寻找的种子资源名", "提示");
                return;
            }

            lstViewOnlinePeer.Items.Clear();
            // 初始化要搜索的资源名
            PeerName searchSeed = new PeerName("0." + tbxSeed.Text
            // PeerNameResolver类是将节点名解析为PeerNameRecord的值（即
            // PeerNameRecord用来定于云中的各个节点
            PeerNameResolver myresolver = new PeerNameResolver();

            // PeerNameRecordCollection表示PeerNameRecord元素的容器
            // Resolve方法是同步的完成解析
            // 使用同步方法可能会出现界面“假死”现象
            // 解决界面假死现象可以采用多线程或异步的方式
            // 关于多线程的知识可以参考本人博客中多线程系列我前面UDP编程中有
            // 在这里就不列出多线程的使用了，朋友可以自己实现，如果有问题可以
            PeerNameRecordCollection recordCollection = myresolver
            foreach (PeerNameRecord record in recordCollection)
            {
                foreach(IPEndPoint endpoint in record.EndPointColle
                {
                    if (endpoint.AddressFamily.Equals(AddressFamily
                    {
                        ListViewItem item = new ListViewItem();
                        item.SubItems.Add(endpoint.ToString());
                        item.SubItems.Add(Encoding.UTF8.GetString(
                        lstViewOnlinePeer.Items.Add(item);
                    }
                }
            }
        }
```

运行结果截图：为了演示资源发现的效果，所以同时开启了本程序的3个进程来模拟网络上对等的3个计算机节点，当在资源名中输入资源后会在分享下列表中显示出本地分享的资源，同时在P2P网络上的其他计算机可以通过资源名称搜索该资源，将得到的资源名称和发布时间显示在ListView控件中，下面是程序的运行结果：



五、总结

到这里P2P编程的介绍就结束了， 本专题只是简单演示了一个资源发现的程序，资源发现是P2P的核心技术，正是因为P2P技术实现了互联网范围的资源发现，才使得它被广泛应用，像我们经常用的下载工具——迅雷，迅雷就是典型采用P2P技术的应用程序，当我们在迅雷页面中输入“爱情公寓3”（相当于本专题中种子文本框填的资源名）然后点击搜索后迅雷会自动启动“狗狗搜索”并显示资源链接列表，当我们点击连接就可以进行下载了。不过迅雷肯定不是使用微软的PNPR，而是迅雷自主研发的与PNPR作用一样的协议——都是完成解析网络资源的地址的作用，当然，迅雷软件中也采用了其他的一些技术，如搜索引擎等。

希望本专题可以帮助大家对P2P技术有所了解，如果有任何的问题都可以通过留言的方式来一起讨论，在下一个专题中将介绍实现一个类似QQ的程序。

源码附上：http://files.cnblogs.com/zhili/P2PResourceDiscovery.zip ，希望觉得有帮助的朋友可以推荐下。谢谢支持

[C# 网络编程系列]专题九：实现类似QQ的即时通信程序

引言：

前面专题中介绍了UDP、TCP和P2P编程，并且通过一些小的示例来让大家更好的理解它们的工作原理以及怎样.Net类库去实现它们的。为了让大家更好的理解我们平常中常见的软件QQ的工作原理，所以在本专题中将利用前面专题介绍的知识来实现一个类似QQ的聊天程序。

一、即时通信系统

在我们的生活中经常使用即时通信的软件，我们经常接触到的有：QQ、阿里旺旺、MSN等等。这些都是属于即时通信（Instant Messenger，IM）软件，IM是指所有能够即时发送和接收互联网消息的软件。

在前面专题P2P编程中介绍过P2P系统分两种类型——单纯型P2P和混合型P2P（QQ就是属于混合型的应用），混合型P2P系统中的服务器（也叫索引服务器）起到协调的作用。在文件共享类应用中，如果采用混合型P2P技术的话，索引服务器就保存着文件信息，这样就可能会造成版权的问题，然而在即时通信类的软件中， 因为客户端传递的都是简单的聊天文本而不是网络媒体资源，这样就不存在版权问题了，在这种情况下，就可以采用混合型P2P技术来实现我们的即时通信软件。前面已经讲了，腾讯的QQ就是属于混合型P2P的软件。

因此本专题要实现一个类似QQ的聊天程序，其中用到的P2P技术是属于混合型P2P，而不是前一专题中的采用的单纯型P2P技术，同时本程序的实现也会用到TCP、UDP编程技术。具体的相关内容大家可以查看本系列的相关专题的。

二、程序实现的详细设计

本程序采用P2P方式，各个客户端之间直接发消息进行聊天，服务器在其中只是起到协调的作用，下面先理清下程序的流程：

2.1 程序流程设计

当一个新用户通过客户端登陆系统后，从服务器获取当在线的用户信息列表，列表信息包括系统中每个用户的地址，然后用户就可以单独向其他发消息。如果有用户加入或者在线用户退出时，服务器就会及时发消息通知系统中的所有其他客户端，达到它们即时地更新用户信息列表。

根据上面大致的描述，我们可以把系统的流程分为下面几步来更好的理解（大家可以参考QQ程序将会更好的理解本程序的流程）：

1. 用户通过客户端进入系统，向服务器发出消息，请求登陆
2. 服务器收到请求后，向客户端返回回应消息，表示同意接受该用户加入，并把自己（指的是服务器）所在监听的端口发送给客户端
3. 客户端根据服务器发送过来的端口号和服务器建立连接
4. 服务器通过该连接 把在线用户的列表信息发送给新加入的客户端。

5. 客户端获得了在线用户列表后就可以自己选择在线用户聊天。（程序中另外设计一个类似QQ的聊天窗口来进行聊天）
6. 当用户退出系统时也要及时通知服务器，服务器再把这个消息转发给每个在线的用户，使客户端及时更新本地的用户信息列表。

2.2 通信协议设计

所谓协议就是约定，即服务器和客户端之间会话信息的内容格式进行约定，使双方都可以识别，达到更好的通信。

下面就具体介绍下协议的设计：

1. 客户端和服务器之间的对话

（1）登陆过程

① 客户端用匿名UDP的方式向服务器发出下面的信息：

login, username, localIPEndPoint

消息内容包括三个字段,每个字段用 “,”分割，login表示的是请求登陆；username表示用户名；localIPEndPint表示客户端本地地址。

② 服务器收到后以匿名UDP返回下面的回应：

Accept, port

其中Accept表示服务器接受请求，port表示服务器所在的端口号，服务器监听着这个端口的客户端连接

③ 连接服务器，获取用户列表

客户端从上一步获得了端口号，然后向该端口发起TCP连接，向服务器索取在线用户列表，服务器接受连接后将用户列表传输到客户端。用户列表信息格式如下：

username1,IPEndPoint1;username2,IPEndPoint2;...;end

username1、username2表示用户名，IPEndPoint1,IPEndPoint2表示对应的端点，每个用户信息都是由"用户名+端点"组成，用户信息以“;”隔开，整个用户列表以“end”结尾。

（2）注销过程

用户退出时，向服务器发送如下消息：

logout,username,localIPEndPoint

这条消息看字面意思大家都知道就是告诉服务器 username+localIPEndPoint这个用户要退出了。

2. 服务器管理用户

（1）新用户加入通知

因为系统中在线的每个用户都有一份当前在线用户表，因此当有新用户登录时，服务器不需要重复地给系统中的每个用户再发送所有用户信息，只需要将新加入用户的信息通知其他用户，其他用户再更新自己的用户列表。

服务器向系统中每个用户广播如下信息：

login,username,remoteIPEndPoint

在这个过程中服务器只是负责将收到的"login"信息转发出去。

（2）用户退出

与新用户加入一样，服务器将用户退出的消息进行广播转发：

logout,username,remoteIPEndPoint

3. 客户端之间聊天

用户进行聊天时，各自的客户端之间是以P2P方式进行工作的，不与服务器有直接联系，这也是P2P技术的特点。

聊天发送的消息格式如下：

talk, longtime, selfUserName, message

其中，talk表明这是聊天内容的消息；longtime是长时间格式的当前系统时间；selfUserName为发送发的用户名；message表示消息的内容。

协议设计介绍完后，下面就进入本程序的具体实现的介绍的。

注：协议是本程序的核心，也是所有软件的核心，每个软件产品的协议都是不一样的，QQ有自己的一套协议，MSN又有另一套协议，所以使用的QQ的用户无法和用MSN的朋友进行聊天。

三、程序的实现

服务器端核心代码：

View Code

```
 1  // 启动服务器
 2          // 根据博客中协议的设计部分
 3          // 客户端先向服务器发送登录请求，然后通过服务器返回的端口号
 4          // 再与服务器建立连接
 5          // 所以启动服务按钮事件中有两个套接字：一个是接收客户端信息套接字
 6          // 监听客户端连接套接字
 7          private void btnStart_Click(object sender, EventArgs e)
 8          {
 9              // 创建接收套接字
10              serverIp = IPAddress.Parse(txbServerIP.Text);
11              serverIPEndPoint = new IPEndPoint(serverIp, int.Par
12              receiveUdpClient = new UdpClient(serverIPEndPoint);
13              // 启动接收线程
14              Thread receiveThread = new Thread(ReceiveMessage);
```

```
15              receiveThread.Start();
16              btnStart.Enabled = false;
17              btnStop.Enabled = true;
18
19              // 随机指定监听端口
20              Random random = new Random();
21              tcpPort = random.Next(port + 1, 65536);
22
23              // 创建监听套接字
24              tcpListener = new TcpListener(serverIp, tcpPort);
25              tcpListener.Start();
26
27              // 启动监听线程
28              Thread listenThread = new Thread(ListenClientConne
29              listenThread.Start();
30              AddItemToListBox(string.Format("服务器线程{0}启动，监
31          }
32
33          // 接收客户端发来的信息
34          private void ReceiveMessage()
35          {
36              IPEndPoint remoteIPEndPoint = new IPEndPoint(IPAdd
37              while (true)
38              {
39                  try
40                  {
41                      // 关闭receiveUdpClient时下面一行代码会产生异常
42                      byte[] receiveBytes = receiveUdpClient.Rece
43                      string message = Encoding.Unicode.GetStrin
44
45                      // 显示消息内容
46                      AddItemToListBox(string.Format("{0}:{1}",re
47
48                      // 处理消息数据
49                      // 根据协议的设计部分，从客户端发送来的消息是具有一
50                      // 服务器接收消息后要对消息做处理
51                      string[] splitstring = message.Split(',');
52                      // 解析用户端地址
53                      string[] splitsubstring = splitstring[2].Sp
54                      IPEndPoint clientIPEndPoint = new IPEndPoi
55                      switch (splitstring[0])
56                      {
57                          // 如果是登录信息，向客户端发送应答消息和广播
58                          case "login":
59                              User user = new User(splitstring[1
60                              // 往在线的用户列表添加新成员
61                              userList.Add(user);
62                              AddItemToListBox(string.Format("用
63                              string sendString = "Accept," + tc
64                              // 向客户端发送应答消息
65                              SendtoClient(user, sendString);
66                              AddItemToListBox(string.Format("向{
67                              for (int i = 0; i < userList.Count
```

```
 68                         {
 69                             if (userList[i].GetName() != us
 70                             {
 71                                 // 给在线的其他用户发送广播消息
 72                                 // 通知有新用户加入
 73                                 SendtoClient(userList[i], r
 74                             }
 75                         }
 76 
 77                         AddItemToListBox(string.Format("广护
 78                         break;
 79                     case "logout":
 80                         for (int i = 0; i < userList.Count;
 81                         {
 82                             if (userList[i].GetName() == sp
 83                             {
 84                                 AddItemToListBox(string.For
 85                                 userList.RemoveAt(i); // 移
 86                             }
 87                         }
 88                         for (int i = 0; i < userList.Count;
 89                         {
 90                             // 广播注销消息
 91                             SendtoClient(userList[i], messa
 92                         }
 93                         AddItemToListBox(string.Format("广护
 94                         break;
 95                 }
 96             }
 97             catch
 98             {
 99                 // 发送异常退出循环
100                 break;
101             }
102         }
103         AddItemToListBox(string.Format("服务线程{0}终止", ser
104     }
105 
106     // 向客户端发送消息
107     private void SendtoClient(User user, string message)
108     {
109         // 匿名方式发送
110         sendUdpClient = new UdpClient(0);
111         byte[] sendBytes = Encoding.Unicode.GetBytes(messag
112         IPEndPoint remoteIPEndPoint =user.GetIPEndPoint();
113         sendUdpClient.Send(sendBytes,sendBytes.Length,remot
114         sendUdpClient.Close();
115     }
116 
117     // 接受客户端的连接
118     private void ListenClientConnect()
119     {
120         TcpClient newClient = null;
```

```
121                 while (true)
122                 {
123                     try
124                     {
125                         newClient = tcpListener.AcceptTcpClient();
126                         AddItemToListBox(string.Format("接受客户端{0
127                     }
128                     catch
129                     {
130                         AddItemToListBox(string.Format("监听线程({0}
131                         break;
132                     }
133 
134                     Thread sendThread = new Thread(SendData);
135                     sendThread.Start(newClient);
136                 }
137             }
138 
139             // 向客户端发送在线用户列表信息
140             // 服务器通过TCP连接把在线用户列表信息发送给客户端
141             private void SendData(object userClient)
142             {
143                 TcpClient newUserClient = (TcpClient)userClient;
144                 userListstring = null;
145                 for (int i = 0; i < userList.Count; i++)
146                 {
147                     userListstring += userList[i].GetName() + ","
148                         + userList[i].GetIPEndPoint().ToString() +
149                 }
150 
151                 userListstring += "end";
152                 networkStream = newUserClient.GetStream();
153                 binaryWriter = new BinaryWriter(networkStream);
154                 binaryWriter.Write(userListstring);
155                 binaryWriter.Flush();
156                 AddItemToListBox(string.Format("向{0}发送[{1}]", new
157                 binaryWriter.Close();
158                 newUserClient.Close();
159             }
```

客户端核心代码：

View Code

```
  1 // 登录服务器
  2             private void btnlogin_Click(object sender, EventArgs e)
  3             {
  4                 // 创建接受套接字
  5                 IPAddress clientIP = IPAddress.Parse(txtLocalIP.Tex
  6                 clientIPEndPoint = new IPEndPoint(clientIP, int.Par
  7                 receiveUdpClient = new UdpClient(clientIPEndPoint);
```

```
 8                // 启动接收线程
 9                Thread receiveThread = new Thread(ReceiveMessage);
10                receiveThread.Start();
11
12                // 匿名发送
13                sendUdpClient = new UdpClient(0);
14                // 启动发送线程
15                Thread sendThread = new Thread(SendMessage);
16                sendThread.Start(string.Format("login,{0},{1}", txt
17
18                btnlogin.Enabled = false;
19                btnLogout.Enabled = true;
20                this.Text = txtusername.Text;
21            }
22
23            // 客户端接受服务器回应消息
24            private void ReceiveMessage()
25            {
26                IPEndPoint remoteIPEndPoint = new IPEndPoint(IPAddr
27                while (true)
28                {
29                    try
30                    {
31                        // 关闭receiveUdpClient时会产生异常
32                        byte[] receiveBytes = receiveUdpClient.Rece
33                        string message = Encoding.Unicode.GetString
34
35                        // 处理消息
36                        string[] splitstring = message.Split(',');
37
38                        switch (splitstring[0])
39                        {
40                            case "Accept":
41                                try
42                                {
43                                    tcpClient = new TcpClient();
44                                    tcpClient.Connect(remoteIPEndPo
45                                    if (tcpClient != null)
46                                    {
47                                        // 表示连接成功
48                                        networkStream = tcpClient.G
49                                        binaryReader = new BinaryRe
50                                    }
51                                }
52                                catch
53                                {
54                                    MessageBox.Show("连接失败", "异常
55                                }
56
57                                Thread getUserListThread = new Thre
58                                getUserListThread.Start();
59                                break;
60                            case "login":
```

```
61                             string userItem = splitstring[1] +
62                             AddItemToListView(userItem);
63                             break;
64                         case "logout":
65                             RemoveItemFromListView(splitstring[
66                             break;
67                         case "talk":
68                             for (int i = 0; i < chatFormList.Co
69                             {
70                                 if (chatFormList[i].Text == spl
71                                 {
72                                     chatFormList[i].ShowTalkInf
73                                 }
74                             }
75 
76                             break;
77                     }
78                 }
79                 catch
80                 {
81                     break;
82                 }
83             }
84         }
85 
86         // 从服务器获取在线用户列表
87         private void GetUserList()
88         {
89             while (true)
90             {
91                 userListstring = null;
92                 try
93                 {
94                     userListstring = binaryReader.ReadString();
95                     if (userListstring.EndsWith("end"))
96                     {
97                         string[] splitstring = userListstring.S
98                         for (int i = 0; i < splitstring.Length
99                         {
100                            AddItemToListView(splitstring[i]);
101                        }
102 
103                        binaryReader.Close();
104                        tcpClient.Close();
105                        break;
106                    }
107                }
108                catch
109                {
110                    break;
111                }
112            }
113        }
```

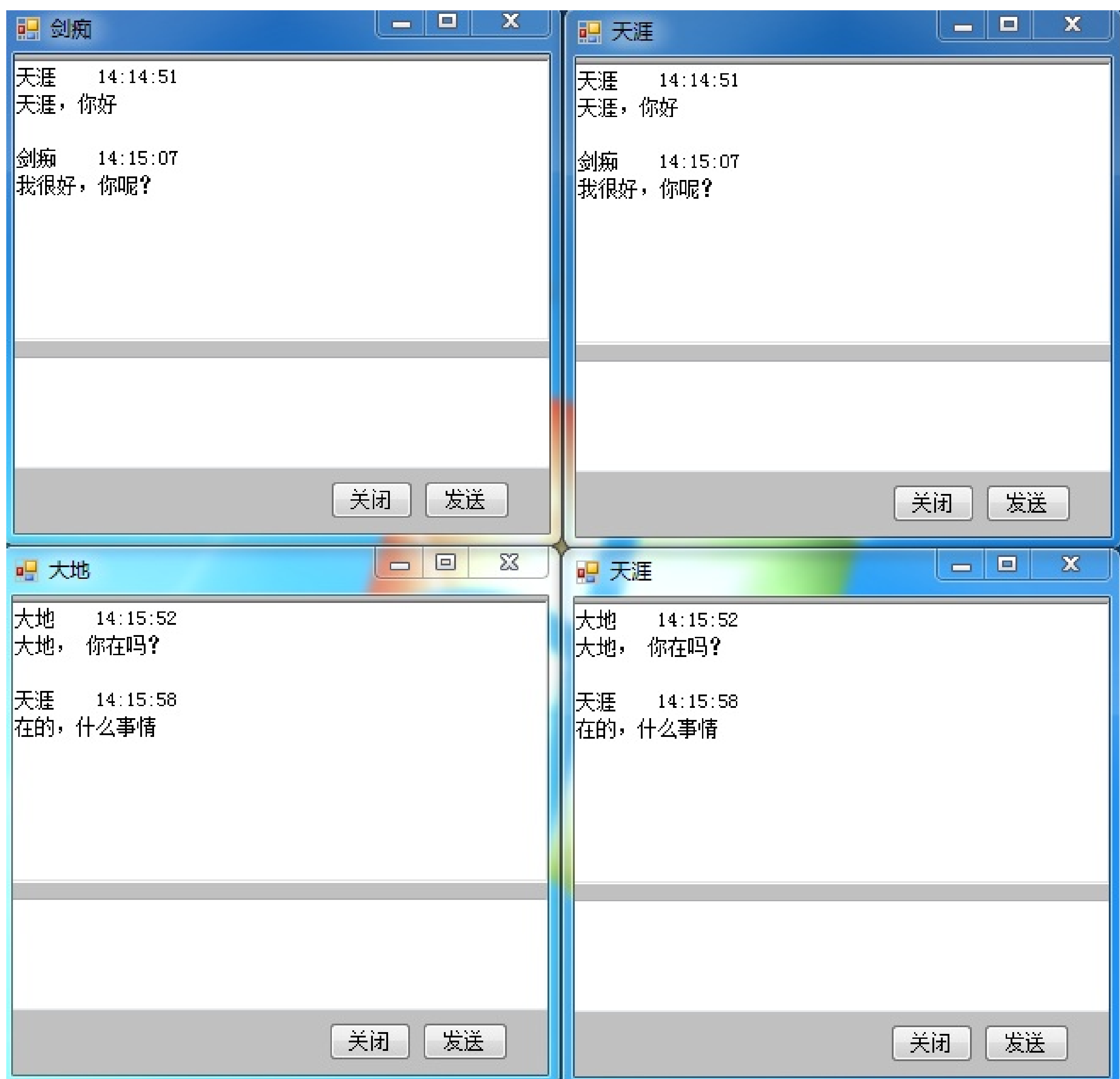
```
114     // 发送登录请求
115             private void SendMessage(object obj)
116             {
117                 string message = (string)obj;
118                 byte[] sendbytes = Encoding.Unicode.GetBytes(messag
119                 IPAddress remoteIp = IPAddress.Parse(txtserverIP.Te
120                 IPEndPoint remoteIPEndPoint = new IPEndPoint(remote
121                 sendUdpClient.Send(sendbytes, sendbytes.Length, rem
122                 sendUdpClient.Close();
123             }
```

程序的运行结果：

首先先运行服务器窗口，在服务器窗口点击“启动”按钮来启动服务器，然后客户端首先指定服务器的端口号，修改用户名（这里也可以不修改，使用默认的也可以），然后点击“登录”按钮来登陆服务器（也就是告诉服务器本地的客户端地址），然后从服务器端获得在线用户列表，界面演示如下：



然后用户可以双击在线用户进行聊天（此程序支持与多人进行聊天），下面是功能的演示图片：



双方进行聊天时，这里没有实现像QQ一样，有人发信息来在对应的客户端就有消息提醒的功能的， 所以双方进行聊天的过程中，每个客户端都需要在在线用户列表中点击聊天的对象来激活聊天对话框（意思就是从图片中可以看出“天涯”客户端想和剑痴聊天的话，就在“在线用户”列表双击剑痴来激活聊天窗口，同时“剑痴”客户端也必须双击“天涯”来激活聊天窗口，这样双方就看到对方发来的信息了，（不激活窗口，也是发送了信息，只是没有一个窗口来进行显示）），而且从图片中也可以看出——此程序支持与多人聊天，即天涯同时与“剑痴”和"大地"同时聊天。

四、总结

本专题介绍了如何去实现一个类似QQ的聊天程序，一方面让大家可以巩固前面专题的内容，另一方面让大家更好的理解即时通信软件（腾讯QQ）的工作原理和软件协议的设计。

后面一专题将介绍如何去实现邮件系统中常用的功能——实现一个简单的邮件应用。

本程序的源代码链接：http://files.cnblogs.com/zhili/IM.zip 觉得有帮助的话还望推荐下，如果有任何意见可以留言。谢谢大家的支持

[C# 网络编程系列]专题十：实现简单的邮件收发器

引言：

在我们的平常工作中，邮件的发送和接收应该是我们经常要使用到的功能的。因此知道电子邮件的应用程序的原理也是非常有必要的，在这一个专题中将介绍电子邮件应用程序的原理、电子邮件应用程序中涉及的协议和实现一个简答的电子邮件收发器程序。

一、邮件应用程序基本知识

1.1 电子邮件原理及相关协议

说到电子邮件的原理，其实和我们现实生活中寄邮件和寄包裹是一样的原理的。就让我们先回顾下现实生活中寄邮件的流程吧——首先，我们先写好信，信封上面写好收信人的地址，写信人的地址，然后把信放到寄信箱中，然后邮局的人会某个时候去这个信箱中的信取出来，然后邮局的人根据信封上写的收信人地址进行转发到当地的邮局，当地邮局然后把信寄到收信人的信箱中（寄包裹的话可能会电话联系，像我们在淘宝，京东买的东西的，收货人就是通过电话联系一样），最后收信人会到自己的信箱中收取信件。上面大致是我们平时生活中寄信的一个流程的。前面已经讲过电子邮件的原理和这个差不多的，下面就介绍了本专题中电子邮件的原理，大家可以和现实生活中的寄信过程进行对比下的，这样可以更加容易理解和掌握：

我们通过电子邮件应用（例如 基于客户端的*Outlook*电子邮件软件 和一些基于*Web*的电子邮件系统——新浪邮箱、谷歌邮箱、*QQ*邮箱等都属于电子邮件应用）将一封写好的邮件（相当于现实生活中的信，当然邮件也要写明收件人地址，邮件内容等信息的）通过电子邮件协议（*SMTP*，在后面的电子邮件相关协议中会介绍）发送到SMTP服务器（就是存储邮件的地方，相当于生活中的邮局一样），然后SMTP服务器根据收件人的地址通过SMTP协议转发到相应SMTP接收服务器上，（*SMTP*服务器进行转发相当于现实生活中邮局的人配送信的过程，配送到收件人当地的邮局，然而现实生活中邮局都是一家，所以可以相互识别——意思就是发送到当地邮局，当地邮局会接收，并且帮助你发送到指定人的信箱中，在网上上就是通过*SMTP*协议来规定这样的一个过程的,发送到别人的*SMTP*服务器上别人的服务器必须要认识发送来的邮件并接收）结束，接收端邮件服务器（POP3服务器）把邮件存放到接受者的电子信箱内（相当于当地邮局的人把信放到收信人的邮箱中），最后收件人可以登录自己的电子信箱，再与POP3服务器进行连接，从POP3服务器上下载发送来的邮件，这样在收件人的电子信箱中就可以看到发送来的电子邮件了（这就是现实生活中收信人从自己的信箱中取信的一个过程）。

注：括号中都是个人的理解，如果有什么不对的地方还望大家指出来，我好及时更正。

上面已经把电子邮件的原理和现实生活中寄信的过程进行对比，相信大家可以更加清楚电子邮件的原理和发送接收过程的，其实网络上的很多应用都可以以现实生活的例子去理解，这样的话我认为可以加深对知识的理解。下面就介绍下电子邮件中的相关协议的内容：

网络上的应用的核心就是协议，因为协议让网络上的客户端相互认识发生来的数据，所以电子邮件应用也不例外，也有相关的电子邮件协议来完成发送电子邮件和接收电子邮件的过程，这些协议主要是：SMTP（简单邮件传输协议，Simple Mail Transfer Protocol）、POP3（邮局协议，Post Office Protocol）和IMAP（网络邮件访问协议，Internet Message Access Protocol）。

- SMTP——SMTP 主要负责将邮件从一台机器转发至另一台机器（可以对照上面电子邮件的过程来理解SMTP的作用）
- POP3——3表示POP协议的版本，主要负责将邮件从邮箱中（POP3服务器）传输到本地计算机。
- IMAP——现在常用的版本为第四版本，即IMAP4，主要负责邮件的检索和处理功能，客户端不需要下载邮件到本地计算机，可直接从邮件客户端软件对服务器上的信件和文 件目录进行操作，它是POP3的替代协议的。

1.2 邮件系统的分类

邮件系统主要分为两类的——基于客户端的邮件系统和基于Web浏览器的邮件系统。Office OutLook就是基于客户端的邮件客户端系统，而像我们经常使用的QQ邮箱、新浪、网易邮箱等都是属于基于Web浏览器的邮件系统，基于客户端的邮件系统的收发过程，通过下面的图片来描述（图片从网上摘下的）：

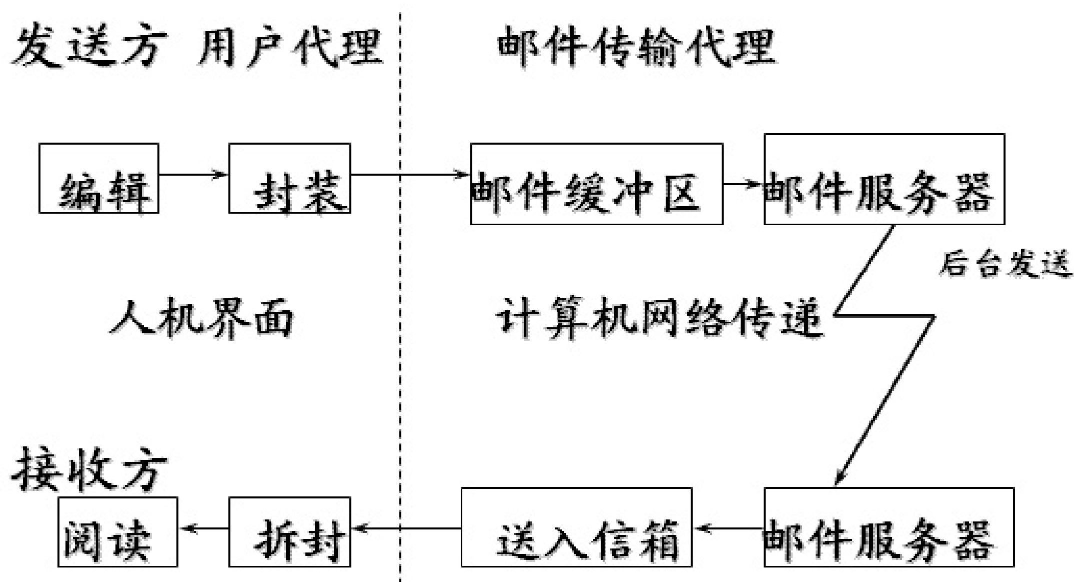


图 1.1 基于客户端的邮件收发过程

发送方通过邮件客户端，将编辑好的邮件向邮件服务（SMTP服务器，在发送过程中也叫发送端邮件服务器）发送，发送端邮件服务器根据收件人的地址来识别接收端邮件服务器（POP3服务器），然后向POP3服务器发送邮件信息，接收端邮件服务器将邮件存放在接收者的电子信箱中，并告知接收者有新邮件，接收者通过邮件客户端与POP3服务器连接后，就可以查看新邮件。

然而，基于Web浏览器的邮件系统与基于客户端的邮件系统不同的地方有：

- 基于Web浏览器邮件系统用户代理(代理的概念也就是用户不是直接与服务器进行通信，而是通过代理的方式，让代理去与服务器通信，然后用户在从代理中获的服务器的信息，代理也就是中间人的作用，相当于生活中中介，在.net中

很多技术都用到了代理，例如委托的概念其实也就是代理的一个概念的)是Web浏览器，基于客户端的邮件系统而是邮件客户端应用程序，一般是Windows Form程序。
- 浏览器发送邮件到SMTP服务器和从POP3服务器中获得邮件的方式都是通过HTTP协议来实现，与基于客户端的邮件系统不同（基于客户端的邮件系统发送通过SMTP协议或ESMTP（Extended SMTP），获得通过POP3或IMAP协议）。

1.3 目前主要的电子邮件服务系统

电子邮件服务系统——就是向大家提供邮箱服务的服务系统，这样的系统当然是由专门的公司进行研发的，我们一般叫这样的公司为邮件服务商，我们平常使用的网易邮箱，新、Gmail邮箱等都是建立在电子邮件服务系统（这里我的理解是——我们使用的新浪，网易等邮箱相当于现实生活中每个人的信箱，通过信箱可以获得邮局来的信，同样道理通过邮箱可以获得邮件服务系统的邮件，这样电子邮件系统相当于邮局）。现在主要电子邮件服务系统主要有下面几种：

- 基于Postfix/Qmail的邮件系统。例如，雅虎邮箱基于Qmail系统
- 微软Exchange 邮件系统
- IBM Lotus Domino邮件系统
- Scalix邮件系统
- Zimbra邮件系统
- MDeamon邮件系统

二、**.Net** 平台对邮件发送功能的支持

在.NET类库中，在System.Net.Mail命名空间下定义了对邮件处理的类，这样使邮件的发送更加方便（这些类也就是对*SMTP*协议的封装，使我们更好地区编程，只需要使用类中的方法和属性等去完成邮件的发送，避免写复杂的*SMTP*协议的命令），下面是一张在System.Net.Mail命名空间下对邮件发送的支持的类截图：

System.Net 命名空间
- System.Net.Mail
  - AlternateView 类
  - AlternateViewCollection 类
  - Attachment 类
  - AttachmentBase 类
  - AttachmentCollection 类
  - DeliveryNotificationOptions 枚举
  - LinkedResource 类
  - LinkedResourceCollection 类
  - MailAddress 类
  - MailAddressCollection 类
  - MailMessage 类
  - MailPriority 枚举
  - SendCompletedEventHandler 委托
  - SmtpAccess 枚举
  - SmtpClient 类
  - SmtpDeliveryFormat 枚举
  - SmtpDeliveryMethod 枚举
  - SmtpException 类
  - SmtpFailedRecipientException 类
  - SmtpFailedRecipientsException 类
  - SmtpPermission 类
  - SmtpPermissionAttribute 类
  - SmtpStatusCode 枚举

从图片中类的名字中也可以看出每个类的作用的，在这里我就不一个介绍的， 大家可以参考MSDN去看每个类的使用，并且我在后面程序的实现部分也会有详细的注释去介绍程序中使用到类的使用。从图中还可以i看出一点——就是只有SMTP的字样，却没有POP3这样的字样的， 这说明.Net类库本身中并没有提供对POP3协议的封装类， 但是我们可以使用Jmail组件来完成从POP3服务器中收取邮件的功能，具体的使用将在后面的邮件收发器程序中邮件的接收部分介绍的。

三、 邮件收发器程序的实现

3.1 邮件发送功能的实现

3.1.1 SMTP协议

SMTP 协议是用于电子邮件的传输的协议， 电子邮件是通过SMTP服务器进行发送的，SMTP服务器的默认端口为25，通常发送邮件有两种方式——一种是不使用客户端认证，即客户端可以使用匿名发送邮件（这种方式叫做SMTP） ；另一种是客户端必须提供用户名和密码认证（这种方式叫做ESMTP，Extended SMTP）目前大部分邮件服务器采用用户名和密码认证的方式。

客户端发送邮件过程为——先通过客户端软件（本程序中的邮件收发器）将邮件发送到SMTP服务器，然后再由SMTP服务器发送到目标SMTP服务器。下面介绍SMTP协议的内容：

SMTP协议总共定义了14个命令，命令由命令码和气候的参数域组成， 不区别大小写的（通过前面专题的讲述可以得出各个协议的命令组成都差不多的）， 下面就简单介绍下5个常用的命令码

名称	解释
HELO或EHLO	发送连接到服务器的命令，EHLO主要用于与ESMTP服务器建立连接时发送的命令
MAIL FROM	指定发件人的邮件地址
Rcpt to	指定收件人的邮件地址
Data	指定邮件正文内容，邮件内容以单独一行 "." 表示接触
Quit	关闭与服务器的连接，然后退出

电子邮件由信封、首部、正文和结束符号4部分组成，下面就具体介绍下这4个部分的内容：

1. 信封

信封包括发信人的邮件地址和接收人的邮件地址，具体对应两条SMTP命令——Mail from: mytest1989@sina.cn(发信人的地址)和Rcpt to: 794170314@qq.com

2. 首部

首部中常用的命令有：

- Subject：<邮件主题>——表示邮件的主题
- Date:<时间>——表示发邮件的时间
- reply-to:<邮件地址>——表示邮件的回复地址
- Content-Type:<邮件类型>——表示邮件包含文本、HTML超文本和附件的类型。
- X-Priority:<邮件优先级>——表示邮件发送的优先级，优先级为3表示为普通邮件；如 X-Priority：3

3. 正文

正文当然指的就是邮件的内容了， 用Data命令指定，首部以一个空行结束，下面就是正文部分

4. 结束符号

邮件以“."结束，

接收方收到SMTP命令之后，会给出一个响应码，每个命令都只有一个响应码，SMTP响应码也是由3位数字组成，后面附加一些文本信息，响应信息的格式为：

响应码<空格>文本信息<回车换行>

客户端发出一条命令后，服务器端返回一个响应，发送者在发送下一条命令前必须等待服务器的响应，成功接收到响应码后才继续发送命令。

附：SMTP常用的响应码：

响应码	解释	响应码	解释
211	系统状态或系统帮助响应	421	服务未就绪，关闭了传输通道
214	帮助信息	501	参数格式错误
220	服务就绪	502	命令不可实现
221	服务关闭传输通道	535	用户验证失败
235	用户验证成功	553	邮箱名不可用，要求的操作未执行
334	等待用户输入验证	554	操作失败
354	开始邮件输入		

3.1.2 邮件的发送过程

第一步：客户端与服务器建立连接（该步中客户端首先发送EHLO local 连接命令，服务器如果返回“220”响应码表示服务器准备就绪了，客户端再继续发送“Auto login”命令，请求登录，服务器收到命令后返回“334”响应码，表示要输入用户名，之后客户端发送用户名命令，等到响应后再发送密码命令，具体在程序的实现中也会有注释。）

第二步：客户端发送邮件的信封

第三步：开始发送邮件数据,（包括邮件首部，正文和结束符号，注：结束符号要单独占一行，表示邮件发送结束）

第四步： 客户端与服务器断开连接。

3.1.3 发送功能的实现代码

相信有了上面的理论解释邮件发送的过程后，实现邮件发送的功能并不难的，并且.net类库中SMTPClient类帮我们封装了SMTP协议，使得我们实现邮件发送功能就不要记住那些具体的命令了， 只需要使用该类中提供的方法来完成邮件的发送（当然你也可以通过发送命令的方式实现，SMTPClient类的方法也是帮我们完成发送命令功能而已的），下面是邮件发送功能的核心代码：

View Code

```
1 #region 邮件发送功能代码
2         // 添加附件
3         private void btnAddFile_Click(object sender, EventArgs
4         {
5             OpenFileDialog openFileDialog = new OpenFileDialog(
6             openFileDialog.CheckFileExists = true;
7             // 只接受有效的文件名
8             openFileDialog.ValidateNames = true;
9             // 允许一次选择多个文件作为附件
```

```
10                  openFileDialog.Multiselect = true;
11                  openFileDialog.Filter = "所有文件(*.*)|*.*";
12                  if (openFileDialog.ShowDialog() != DialogResult.OK
13                  {
14                      return;
15                  }
16                  if (openFileDialog.FileNames.Length > 0)
17                  {
18                      // 因为这里允许选择多个文件，所以这里用AddRange而没有
19                      cmbAttachment.Items.AddRange(openFileDialog.Fi
20                  }
21              }
22
23              // 删除附件
24              private void btnDeleteFile_Click(object sender, EventA
25              {
26                  int index = cmbAttachment.SelectedIndex;
27                  if (index == -1)
28                  {
29                      MessageBox.Show("请选择要删除的附件！", "提示", Mes
30                      return;
31                  }
32                  else
33                  {
34                      cmbAttachment.Items.RemoveAt(index);
35                  }
36              }
37
38              // 发送邮件
39              private void btnSend_Click(object sender, EventArgs e)
40              {
41                  this.Cursor = Cursors.WaitCursor;
42                  // 实例化一个发送的邮件
43                  // 相当于与现实生活中先写信，程序中把信（邮件）抽象为邮件类
44                  MailMessage mailMessage = new MailMessage();
45                  // 指明邮件发送的地址，主题，内容等信息
46                  // 发信人的地址为登录收发器的地址，这个收发器相当于我们平时W
47                  mailMessage.From = new MailAddress(tbxUserMail.Text
48                  mailMessage.To.Add(txbSendTo.Text);
49                  mailMessage.Subject = txbSubject.Text;
50                  mailMessage.SubjectEncoding = Encoding.Default;
51                  mailMessage.Body = richtbxBody.Text;
52                  mailMessage.BodyEncoding = Encoding.Default;
53                  // 设置邮件正文不是Html格式的内容
54                  mailMessage.IsBodyHtml = false;
55                  // 设置邮件的优先级为普通优先级
56                  mailMessage.Priority = MailPriority.Normal;
57                  //mailMessage.ReplyTo = new MailAddress(tbxUserMail
58
59                  // 封装发送的附件
60                  System.Net.Mail.Attachment attachment = null;
61                  if (cmbAttachment.Items.Count > 0)
62                  {
```

```
63                  for (int i = 0; i < cmbAttachment.Items.Count;
64                  {
65                      string fileNamePath = cmbAttachment.Items[i
66                      string extName = Path.GetExtension(fileName
67                      if (extName == ".rar" || extName == ".zip")
68                      {
69                          attachment = new System.Net.Mail.Attach
70                      }
71                      else
72                      {
73                          attachment = new System.Net.Mail.Attach
74                      }
75
76                      // 表示MIMEContent-Disposition标头信息
77                      // 对于ContentDisposition具体类的解释大家可以参
78                      // 这里我就不重复贴出来了，给个地址: http://msdn
79                      ContentDisposition cd = attachment.ContentD
80                      cd.CreationDate = File.GetCreationTime(file

81                      cd.ModificationDate = File.GetLastWriteTime
82                      cd.ReadDate = File.GetLastAccessTime(fileNa
83                      // 把附件对象加入到邮件附件集合中
84                      mailMessage.Attachments.Add(attachment);
85                  }
86              }
87
88              // 发送写好的邮件
89              try
90              {
91                  // SmtpClient类用于将邮件发送到SMTP服务器
92                  // 该类封装了SMTP协议的实现，
93                  // 通过该类可以简化发送邮件的过程，只需要调用该类的Send
94                  smtpClient.Send(mailMessage);
95                  MessageBox.Show("邮件发送成功！", "提示", MessageB
96
97              }
98              catch(SmtpException smtpError)
99              {
100                 MessageBox.Show("邮件发送失败：[" + smtpError.Sta
101                     + smtpError.Message+"];\r\n["+smtpError.Sta
102                     ,"错误",MessageBoxButtons.RetryCancel,Messa
103             }
104             finally
105             {
106                 mailMessage.Dispose();
107                 this.Cursor = Cursors.Default;
108             }
109         }
110
111         #endregion
```

3.2 邮件接收功能的实现

3.2.1 POP3协议

前面介绍了邮件的发送，当然接收者需要登录邮箱来查看收到的邮件了，此时就必有有一个协议去读取服务器上邮件，POP3就是这样的一个协议。还有两外一种协议也是用来接收邮件的——IMAP协议，它与POP3协议区别有：1. IMAP使用的端口号是143而POP3邮件服务器通过监听110端口来提供POP3服务；

2 . IMAP 允许客户端在邮件服务器上建立文件夹来保持邮件，而不用把邮件下载到本地。而POP3需要把邮件下载到本地。

和SMTP协议一样，客户端要通过POP3协议从POP3服务器上获取邮件，也需要先与POP3服务器建立TCP连接，等待服务器向客户端发送确认信息表明连接成功时，客户端才可以继续发送命令给服务器来获取邮件，在POP3协议中，规定的命令也是几十条的，每条命令由命令和参数两部分组成，都是以回车换行结束，并且命令和参数之间由空格分隔，命令通常也是由3-4个字母组成，参数最多可以为40个字符的长度，而服务器的响应信息是由一个状态码和可能附加信息的字符组成，所有的响应信息也是以回车换行结束的。状态码和其他协议定义的状态码有点不一样，POP3服务器响应的状态码有两种——“+OK”(确定)和"-ERR"（失败）。这样客户端可以通过检查响应的状态码所包含的字符来判断服务器是否响应客户端发送的命令，即响应信息中包含“+OK”表示成功响应，包含“-ERR”表示服务器未响应。同时在程序的实现中大家可以通过Debug来查看响应消息的组成，这样可以加深理解。

3.2.2 邮件接收的过程

客户端从服务器接收邮件的过程主要经历3个状态：授权状态、操作状态和更新状态

（1）授权状态——客户端发送与POP3服务器的TCP连接请求，服务器接收后发送一个响应确认信息之后，此时客户端需要发送正确的用户名和密码进行确认，因为在邮件服务器上有很多用户邮箱，只有提供正确的用户名和密码才有权限访问自己的邮箱，就像现实生活中我们邮箱的钥匙一样的。

发送用户名命令： USER mytest1989@sina.cn

发送密码命令： PASS **（这两个命令都在代码中有给出的，大家可以参考代码来理解邮件的接收过程）

（2） 操作状态——如果客户端提供了正确的用户名和密码，则授权状态也就通过了，就相当于打开了在服务器上自己的邮箱，现在用户就有权限进去下载，查看和删除邮件等操作的，然后在现实生活中的取邮件和删除邮件都很简单（只要打开了邮箱门，用手去拿就可以了），然后在网络应用上，这些操作都需要发送POP3命令给服务器，服务器接收到命令后再给出响应。操作中常用的命令有：

- STAT 命令——该命令从服务器中获取邮件总数和总字节数，服务器响应命令返回邮件总数和总字节数

如：

```
客户端发送POP3命令： STAT
服务器响应命令： +OK 2 1340
服务器响应命令：
```

- LIST 命令——该命令从服务器中获得邮件列表和大小，服务器响应命令返回列出邮件列表和大小。

如：

```
客户端发送POP3命令：LIST
服务器响应命令： +OK 2 message(1430 octect)
服务器响应命令：1    700
服务器响应命令：2    730
服务器响应命令：<一个空行>
```

- RETR 命令—— 该命令从服务器中获得一个邮件，格式为 RETR <邮件编号>

如：

```
客户端发送POP3命令：RETR 1
服务器响应命令： 700 octets
服务器响应命令：<邮件头和内容>
服务器响应命令： <空行>
```

- DELETE 命令——该命令告诉服务器将邮件标记为删除。（此时只是逻辑删除）

（3）更新状态——客户端发送QUIT命令后，此时就进入更新状态，POP3服务器释放在操作状态中取得的资源，并将逻辑删除的邮件进行物理删除，然后关闭与客户端的TCP连接。这样整个邮件处理的过程就结束了。

3.2.3 接收功能的实现代码

有了前面接收邮件过程的介绍，再参考代码的实现，相信大家可以更好的理解客户端从POP3服务器中获取邮件的过程的，由于.net类库并没有帮我们封装POP3协议的实现类，要实现邮件的获取可以采用发送命令的方式，也可以使用Jmail组件，这个组件其实就是POP3协议的封装类，既然微软没有帮我们做，其他公司帮我们做好后来帮助我们简单的实现邮件的接收的一个类库罢了。然后在使用这个组件的过程中出现了好几个问题的，在源码中我都解释，大家可以下载源代码后查看的。

实现邮件接收的核心代码如下：

View Code

```
 // 登录邮箱（这里是本程序—邮件收发器）
```

```
        private void btnLogin_Click_1(object sender, EventArgs e)
        {
            // 与POP3服务器建立TCP连接
            // 建立连接后把服务器上的邮件下载到本地
            // 设置当前界面的光标为等待光标（就是我们看到的一个动的圆形）
            Cursor.Current = Cursors.WaitCursor;

            lsttbxStatus.Items.Clear();
            try
            {
                // POP3服务器通过监听TCP110端口来提供POP3服务的
                // 向POP3服务器发出tcp请求
                tcpClient = new TcpClient(tbxPOP3Server.Text, 110);
                lsttbxStatus.Items.Add("正在连接...");
            }
            catch
            {
                MessageBox.Show("连接失败", "错误", MessageBoxButtons
                lsttbxStatus.Items.Add("连接失败！");
                return;
            }

            // 连接成功的情况
            networkStream = tcpClient.GetStream();
            streamReader = new StreamReader(networkStream, Encoding
            streamWriter = new StreamWriter(networkStream, Encoding
            streamWriter.AutoFlush = true;
            string str;
            // 读取服务器返回的响应连接信息
            str = GetResponse();
            if (CheckResponse(str) == false)
            {
                lsttbxStatus.Items.Add("服务器拒接了连接请求");
                return;
            }
            // 如果服务器接收请求
            // 向服务器发送凭证——用户名和密码

            // 向服务器发送用户名，请求确认
            lsttbxStatus.Items.Add("核实用户名阶段...");
            SendToServer("USER " + tbxUserMail.Text);
            str = GetResponse();
            if (CheckResponse(str) == false)
            {
                lsttbxStatus.Items.Add("用户名错误.");
                return;
            }

            // 用户名审核通过后再发送密码等待确认
            // 向服务器发送密码，请求确认
            SendToServer("PASS "+txbPassword.Text);
            str = GetResponse();
            if (CheckResponse(str) == false)
```

```
            {
                lsttbxStatus.Items.Add("密码错误！");
                return;
            }

            lsttbxStatus.Items.Add("身份验证成功，可以开始会话");
            // 向服务器发送LIST 命令，请求获得邮件列表和大小
            lsttbxStatus.Items.Add("获取邮件....");
            SendToServer("LIST");
            str = GetResponse();
            if (CheckResponse(str) == false)
            {
                lsttbxStatus.Items.Add("获取邮件列表失败");
                return;
            }

            lsttbxStatus.Items.Add("邮件获取成功");

            // 窗口控件控制
            tabControlMyMailbox.Enabled = true;
            btnReadMail.Enabled = false;
            btnDownLoad.Enabled = false;
            btnDeleteMail.Enabled = false;

            // 登陆成功后实例化邮件发送对象，以便后面完成发送邮件的操作
            // 实例化邮件发送类（SmtpClient）对象
            if (smtpClient == null)
            {
                smtpClient = new SmtpClient();
                smtpClient.Host = tbxSmtpServer.Text;
                smtpClient.Port = 25;

                // 不使用默认凭证，即需要认证登陆
                smtpClient.UseDefaultCredentials = false;
                smtpClient.Credentials = new NetworkCredential(tbxU
                smtpClient.DeliveryMethod = SmtpDeliveryMethod.Netw
            }

            // 登陆成功后，自动接收新邮件
            // 开始接收邮件
            try
            {
                btnRefreshMailList.PerformClick();
            }
            catch
            {
                MessageBox.Show("读取邮件列表失败！", "错误", MessageB
            }

            lsttbxStatus.Items.Add("登陆成功!");
            lsttbxStatus.TopIndex = lsttbxStatus.Items.Count - 1;
            Cursor.Current = Cursors.Default;
```

```
            // 窗口控件控制
            richtbxMailContentReview.Enabled = true;
            tbxUserMail.Enabled = false;
            txbPassword.Enabled = false;
            btnLogin.Enabled = false;
            btnLogout.Enabled = true;
            tbxSmtpServer.Enabled = false;
            tbxPOP3Server.Enabled = false;
            btnReadMail.Enabled = true;
            btnDownLoad.Enabled = true;
            btnDeleteMail.Enabled = true;
            tabControlMyMailbox.Focus();
        }

        #region 处理与POP3服务器交互事件
        // 获取服务器响应的信息
        private string GetResponse()
        {
            string str = null;
            try
            {
                str = streamReader.ReadLine();
                if (str == null)
                {
                    lsttbxStatus.Items.Add("连接失败——服务器没有响应")
                }
                else
                {
                    lsttbxStatus.Items.Add("收到：[" + str + "]");
                    if (str.StartsWith("-ERR"))
                    {
                        str = null;
                    }
                }
            }
            catch(Exception err)
            {
                lsttbxStatus.Items.Add("连接失败：[" + err.Message +
            }

            return str;
        }

        // 检查响应信息
        private bool CheckResponse(string responseString)
        {
            if (responseString == null)
            {
                return false;
            }
            else
            {
                if (responseString.StartsWith("+OK"))
```

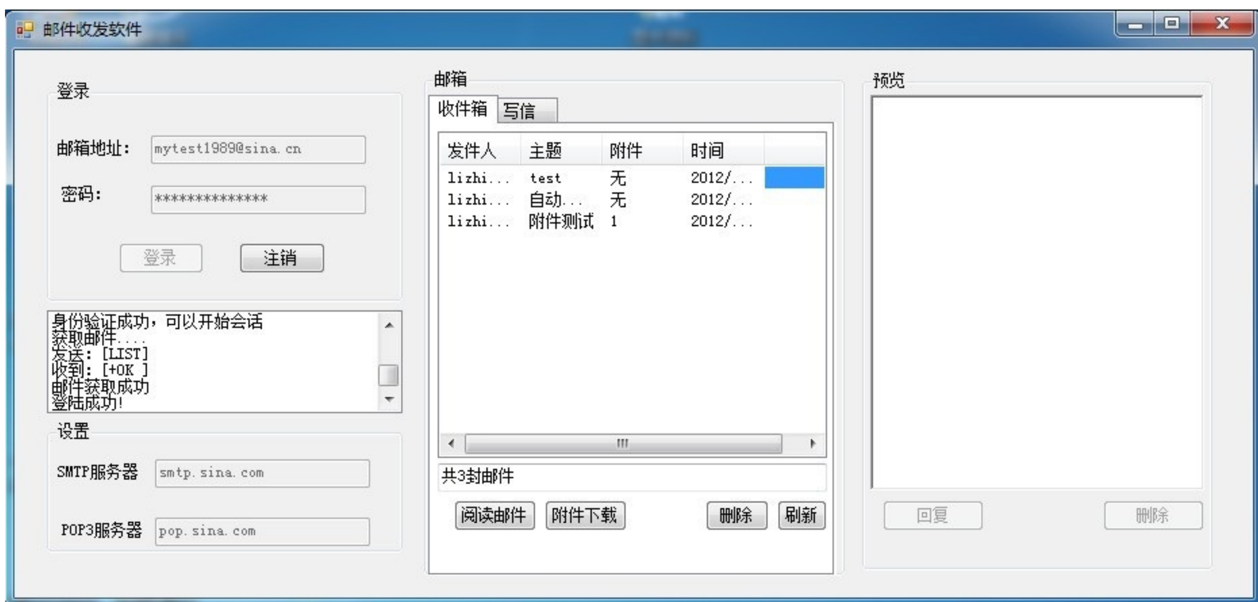
```
                {
                    return true;
                }
                else
                {
                    return false;
                }
            }
        }

        // 向服务器发送命令
        private bool SendToServer(string str)
        {
            try
            {
                // 这里必须使用WriteLine方法的，因为POP3协议中定义的命令是
                // 如果客户端发送的命令没有以回车换行结束，POP3服务器就不能
                // 如果想用Write方法，则str输入的参数字符中必须包含“\r\n”
                streamWriter.WriteLine(str);
                streamWriter.Flush();
                lsttbxStatus.Items.Add("发送：[" + str + "]");
                return true;
            }
            catch(Exception ex)
            {
                lsttbxStatus.Items.Add("发送失败：[" + ex.Message +
                return false;
            }
        }

        #endregion
```

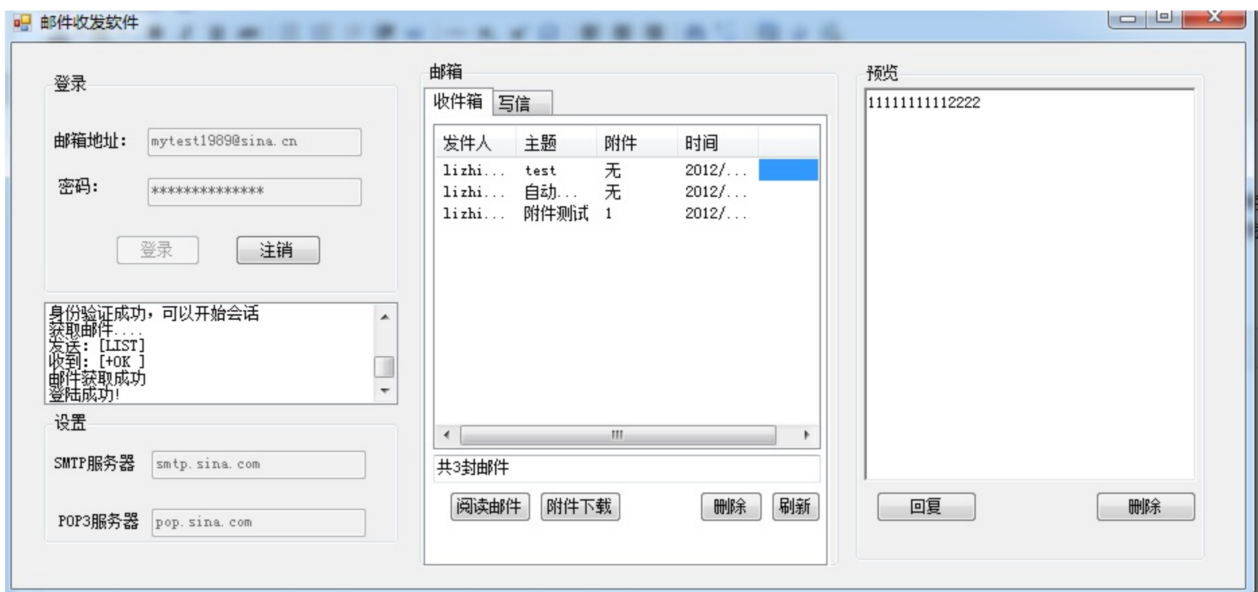
3.3 程序运行结果演示

首先输入邮箱名和密码登录到POP3服务器来获取邮件列表的演示：

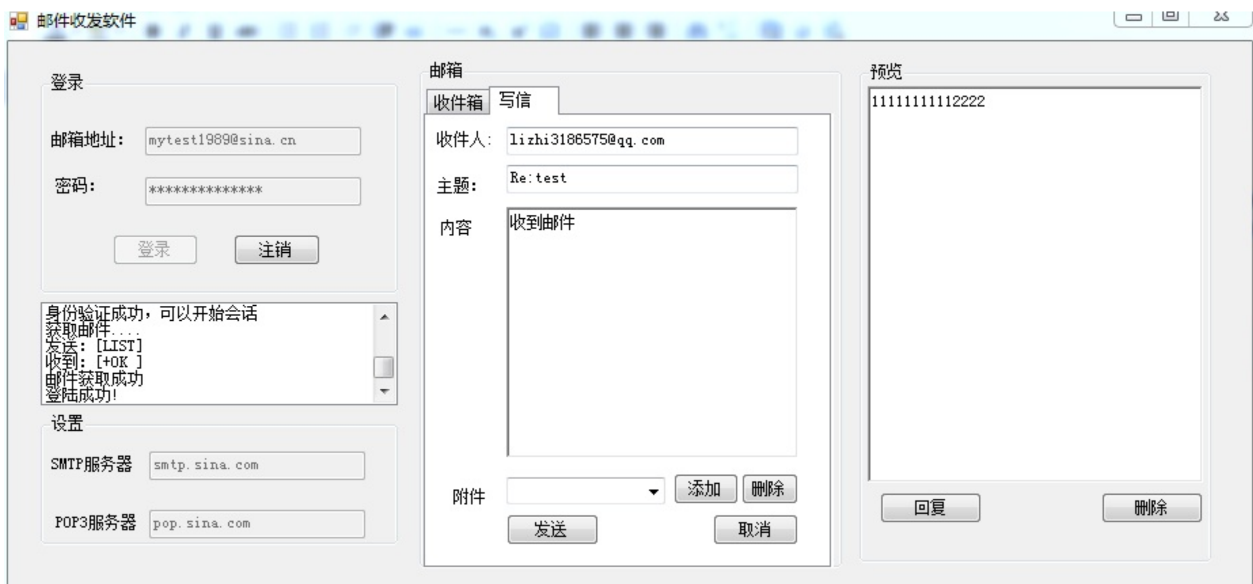


然后在邮件列表中选中一个邮件进行阅读，然后进行回复邮件的操作演示（邮件的发送都可以附加附件发送出去）：

阅读邮件的界面：

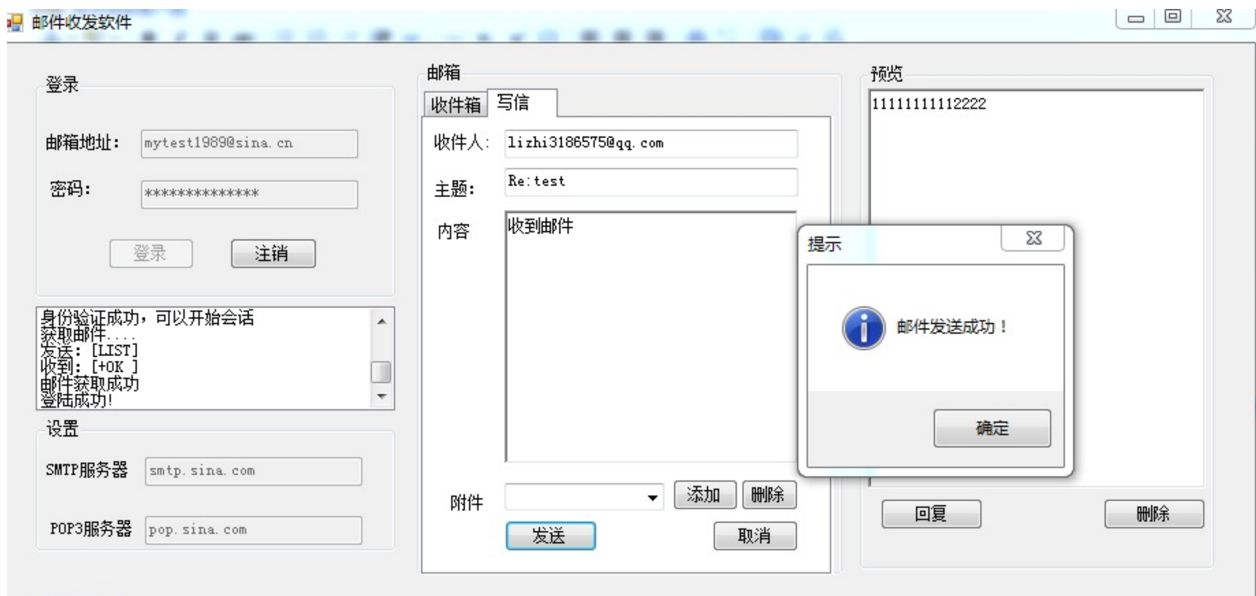


回复邮件的界面：



同时点击发送按钮后，就可以把邮件发送到sina的SMTP服务器上，再由新浪的SMTP服务器转发到QQ的SMTP服务器，QQ的POP3服务器中QQ的SMTP服务器获取收到的邮件，当QQ用户输入正确的邮箱名和密码后就可以从QQ的POP3服务器上获取收到的邮件。

点击发送按钮后成功发送邮件的图片：



四、总结

介绍到这里，本专题的内容就已经介绍完了，希望通过本专题可以让大家明白邮件发送和接收的原理，并且可以自定义一个简单邮件收发器的功能的，在后面一专题将介绍FTP协议（文件传输协议），并实现一个简单的文件上传和下载的程序。

源代码下载地址：http://files.cnblogs.com/zhili/MailSendAndReceive.zip，如果觉得有帮助的话，还望大家推荐下，谢谢大家的支持

[C# 网络编程系列]专题十一：实现一个基于FTP协议的程序——文件上传下载器

引言：

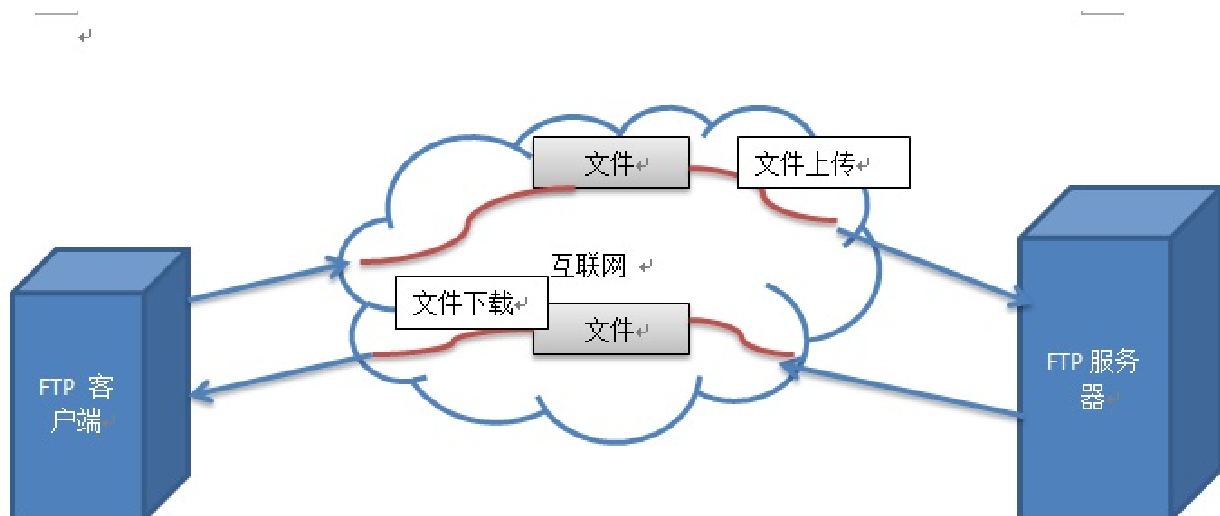
在这个专题将为大家揭开下FTP这个协议的面纱，其实学习知识和生活中的例子都是很相通的，就拿这个专题来说，要了解FTP协议然后根据FTP协议实现一个文件下载器，就和和追MM是差不多的过程的，相信大家追MM都有自己的经验的，我感觉大部分的过程肯定是——第一步：先通过工作关系或者朋友关系等认识MM（认识FTP协议，知道FTP协议的是什么）；第二步：当然了解MM有兴趣爱好了（了解FTP协议有哪些命令和工作过程）第三步：如果对方是你的菜的话，那当然要采取追求的了（就好比用了解到的FTP协议来实现一个文件上传下载器）。不过追MM好像对我来说还是比较难的了，所以还是言归正传了，还是好好的学习我的代码吧，回到本章的内容——FTP的协议。

（注：最近想好好改进下文章的幽默程度，所以文章中会尽量以有趣的生活中的例子来表述网络编程的知识，希望可以让大家在学习知识的同时也可以获得乐趣，如果有什么地方理解不准确的还望大家多多指出。）

一、FTP协议的自我介绍

我们在上学的时候，同学们第一次开学的时候老师一般会组织大学到讲台上进行自我介绍，让同学都互相认识，同样，如果对于没有接触过FTP协议的朋友来说，FTP协议的自我介绍当然也是不可避免的了，这样我们才能进一步去了解FTP协议"这位同学"了，之后才能和他成为好朋友，或者是好基友了。下面就开始FTP协议同学的自我介绍了，大家热烈欢迎。

FTP 协议同学：大家好，我的名字叫FTP，FTP是文件传输协议（File Transfer Protocol,FTP）,我的工作就是负责将文件从一台计算机传输到另外一台计算机，并且我还可以保证传输的可靠性。我的工作流程可以通过下面的一张图来表达：



从图中大家应该可以明白我的工作过程了吧，我的工作过程是典型的C/S模式——我的客户端（在本章实现的文件上传下载器属于客户端）首先发起与我的服务器连接连接，告诉我的服务器说“我现在想和你聊聊天”， 然后我的服务器收到这个请求后给出回答——“聊天，当然可以了，我批准了”，客户端收到这个信息后，就可以服务器之间就建立一条马路或者是通道，然后我的客户端好还想进一步了解下我的服务器，在发出一个说“我想要下载你上面的东西 或者是 我想上传一些文件到你那里，想让你帮我保管下，这样我可以随处都可以从你那里得到我上传的资料的”，我的服务器收到请求后，如果允许客户端这么做的话就会回答说 “可以啊”（就像我们追女生一样，建立好关系后，当然就要表白了，此时我们就说“我很喜欢你之类的话”，然后等待MM的回答，“可以啊” 这个答案都是我们希望听到的答案的），我的客户端听到后非常开心，马上选择自己需要上传的文件或者想从服务器下载的文件找到，上传或者下载该文件的。 我还要补充一点，在访问我的FTP服务器之前必须登录，这样我的服务器才认识你，才可能会搭理你的，登录时就需要客户端提供一个用户名和密码，提供了正确的用户名和密码后就可以和我的服务器进行聊天 和请求上传或下载我服务器上的文件了；然而我的某些服务器提供了一种匿名的方式，我的客户端不需要提供用户名和密码就可以进行聊天了，其实匿名的方式和我聊天的本质是：提供服务的公司或机构在我的服务器上建立一个公用的账户，方便那些没有提供用户名和密码的客户端与我聊天。 上面就是我的自我介绍了，谢谢大家。

二、.Net 为实现我的客户端提供了些什么？

可以说微软真是一位雷锋叔叔的，因为在他的.Net类库中提供了很多类库供我们使用，当然为实现我的客户端也提供了一些类的支持的， 现在就看看这位好人帮我们提供了哪些类来对实现一个FTP客户端程序的支持的。

这位好人通过命名空间System.Net下的FtpWebRequest类和FtpWebResponse类提供对实现FTP客户端的支持。

2.1 FtpWebRequest类

该类是WebRequest类的派生类，FTPWebRequest类用于向服务器发出请求，告诉服务器说“我想和你聊天"，如果要获得FtpWebRequest的一个实例，则需要使用**Create**方法来创建实例，对于该类如何使用我在这里也就不一一列出来的， 大家可以查看MSDN的相关文档来了解方法的使用，并且在本专题实现的程序中也会有所介绍的，下面给出MSDN中的一个链接的：

http://msdn.microsoft.com/zh-cn/library/8exfzxft.aspx

2.2 FtpWebResponse类

FTP客户端既然发话了，服务器当然也要有所表示的了， 不要哑巴一样不说话的，总要给个答复的，FtpWebResponse类就负责封装FTP服务器对客户端请求的回答的一个类。FTP客户端通过**GetResponse**方法来获得FtpWebResponse类的对象的，如果服务器回答说“我们可以聊天的”，这样就说明他们俩就可以互相沟通了，就好比追MM的时候你问MM说“可以给电话号码给我吗？”，然后MM对你也有好感就告诉你一个号码后，得到MM的号码也就和MM建立了沟通的通道了，就好比服务器回答“我们可以聊天的”。之后客户端和服务器就可以进行进一步的沟通（上传文件到服务器或者要求服务器给些文件给客户端），之后的过程就好比你可以通过电话号码和MM进一步的交流，知道MM的有些什么性格和爱好的。下面提供一个

MSDN中该类的使用链接，这里我就不一一介绍他的成员了，大家可以到MSDN中查看的，上面每个属性和方法都有一个比较好的解释，并且大家也可以通过下面实现的FTP客户端程序进一步了解该类的使用：

http://msdn.microsoft.com/zh-cn/library/system.net.ftpwebresponse.aspx

三、如何实现一个**FTP**客户端程序？——看完下面的介绍你就会知道了

通过FTP协议的自我介绍部分大家应该可以明白了FTP协议的工作过程的，然而一个FTP客户端程序就是基于FTP协议的文件上传下载器，通过这个程序大家可以对FTP服务器上的资料进行浏览、上传和下载等操作的。

程序中主要模块的代码：

登录模块：

View Code

```
#region 登录模块的实现
      // 登录服务器事件
      private void btnlogin_Click(object sender, EventArgs e)
      {
          if (tbxServerIp.Text == string.Empty)
          {
              MessageBox.Show("请先填写服务器IP地址", "提示");
              return;
          }

          ftpUristring = "ftp://" + tbxServerIp.Text;
          networkCredential = new NetworkCredential(tbxUsername.T
          if (ShowFtpFileAndDirectory() == true)
          {
              btnlogin.Enabled = false;
              btnlogout.Enabled = true;
              lstbxFtpResources.Enabled = true;
              lstbxFtpState.Enabled = true;
              tbxServerIp.Enabled = false;
              if (chkbxAnonymous.Checked == false)
              {
                  tbxUsername.Enabled = false;
                  tbxPassword.Enabled = false;
                  chkbxAnonymous.Enabled = false;
              }
              else
              {
                  chkbxAnonymous.Enabled = false;
              }

              tbxloginmessage.Text = "登录成功";
              btnUpload.Enabled = true;
              btndownload.Enabled = true;
              btnDelete.Enabled = true;
          }
```

```
            else
            {
                lstbxFtpState.Enabled = true;
                tbxloginmessage.Text = "登录失败";
            }
        }

        // 显示资源列表
        private bool ShowFtpFileAndDirectory()
        {
            lstbxFtpResources.Items.Clear();
            string uri = string.Empty;
            if (currentDir == "/")
            {
                uri = ftpUristring;
            }
            else
            {
                uri = ftpUristring + currentDir;
            }

            string[] urifield = uri.Split(' ');
            uri = urifield[0];
            FtpWebRequest request = CreateFtpWebRequest(uri, WebRec

            // 获得服务器返回的响应信息
            FtpWebResponse response = GetFtpResponse(request);
            if (response == null)
            {
                return false;
            }
            lstbxFtpState.Items.Add("连接成功，服务器返回的是："+respon

            // 读取网络流数据
            Stream stream = response.GetResponseStream();
            StreamReader streamReader = new StreamReader(stream,Enc
            lstbxFtpState.Items.Add("获取响应流....");
            string s = streamReader.ReadToEnd();
            streamReader.Close();
            stream.Close();
            response.Close();
            lstbxFtpState.Items.Add("传输完成");

            // 处理并显示文件目录列表
            string[] ftpdir = s.Split(Environment.NewLine.ToCharArr
            lstbxFtpResources.Items.Add("↑返回上层目录");
            int length = 0;
            for (int i = 0; i < ftpdir.Length; i++)
            {
                if (ftpdir[i].EndsWith("."))
                {
                    length = ftpdir[i].Length - 2;
                    break;
```

```
                }
            }

            for (int i = 0; i < ftpdir.Length; i++)
            {
                s = ftpdir[i];
                int index = s.LastIndexOf('\t');
                if (index == -1)
                {
                    if (length < s.Length)
                    {
                        index = length;
                    }
                    else
                    {
                        continue;
                    }
                }

                string name = s.Substring(index + 1);
                if (name == "." || name == "..")
                {
                    continue;
                }

                // 判断是否为目录，在名称前加"目录"来表示
                if (s[0] == 'd' || (s.ToLower()).Contains("<dir>"))
                {
                    string[] namefield = name.Split(' ');
                    int namefieldlength = namefield.Length;
                    string dirname;
                    dirname = namefield[namefieldlength - 1];

                    // 对齐
                    dirname = dirname.PadRight(34,' ');
                    name = dirname;
                    // 显示目录
                    lstbxFtpResources.Items.Add("[目录]" + name);
                }
            }

            for (int i = 0; i < ftpdir.Length; i++)
            {
                s = ftpdir[i];
                int index = s.LastIndexOf('\t');
                if (index == -1)
                {
                    if (length < s.Length)
                    {
                        index = length;
                    }
                    else
                    {
```

```
                    continue;
                }
            }

            string name = s.Substring(index + 1);
            if (name == "." || name == "..")
            {
                continue;
            }

            // 判断是否为文件
            if (!(s[0] == 'd' || (s.ToLower()).Contains("<dir>'
            {
                string[] namefield = name.Split(' ');
                int namefieldlength = namefield.Length;
                string filename;

                filename = namefield[namefieldlength - 1];

                // 对齐
                filename = filename.PadRight(34, ' ');
                name = filename;

                // 显示文件
                lstbxFtpResources.Items.Add(name);
            }
        }

        return true;
    }

    // 注销事件
    private void btnlogout_Click(object sender, EventArgs e)
    {
        btnlogin.Enabled = true;
        btnlogout.Enabled = false;
        tbxServerIp.Enabled = true;
        tbxServerIp.SelectAll();
        tbxServerIp.Focus();
        chkbxAnonymous.Enabled = true;
        if (chkbxAnonymous.Checked == false)
        {
            tbxUsername.Enabled = true;
            tbxPassword.Enabled = true;
        }

        tbxloginmessage.Text = "你已经退出了。";
        lstbxFtpResources.Items.Clear();
        lstbxFtpResources.Enabled = false;
        lstbxFtpState.Items.Clear();
        lstbxFtpState.Enabled = false;
        btnUpload.Enabled = false;
        btndownload.Enabled = false;
```

```
            btnDelete.Enabled = false;
        }

        #endregion
```

对FTP服务器操作模块（本程序中实现下载、上传和删除的功能）：

View Code

```
    #region 对文件的操作模块实现
        // 上传文件到服务器事件
        private void btnUpload_Click(object sender, EventArgs e)
        {
            // 选择要上传的文件
            OpenFileDialog openFileDialog = new OpenFileDialog();
            openFileDialog.FileName = openFileDialog.FileNames.ToSt
            openFileDialog.Filter = "所有文件(*.*)|*.*";
            if (openFileDialog.ShowDialog() != DialogResult.OK)
            {
                return;
            }

            FileInfo fileinfo = new FileInfo(openFileDialog.FileNar
            try
            {
                string uri = GetUriString(fileinfo.Name);
                FtpWebRequest request = CreateFtpWebRequest(uri, We
                request.ContentLength = fileinfo.Length;
                int buflength = 8196;
                byte[] buffer = new byte[buflength];
                FileStream filestream = fileinfo.OpenRead();
                Stream responseStream = request.GetRequestStream();
                lstbxFtpState.Items.Add("打开上传流，文件上传中...");
                int contenlength = filestream.Read(buffer, 0, bufle
                while (contenlength != 0)
                {
                    responseStream.Write(buffer, 0, contenlength);
                    contenlength = filestream.Read(buffer, 0, bufle
                }

                responseStream.Close();
                filestream.Close();
                FtpWebResponse response = GetFtpResponse(request);
                if (response == null)
                {
                    lstbxFtpState.Items.Add("服务器未响应...");
                    lstbxFtpState.TopIndex = lstbxFtpState.Items.Co
                    return;
                }

                lstbxFtpState.Items.Add("上传完毕，服务器返回：" + res
```

```
                lstbxFtpState.TopIndex = lstbxFtpState.Items.Count
                MessageBox.Show("上传成功！");

                // 上传成功后，立即刷新服务器目录列表
                ShowFtpFileAndDirectory();
            }
            catch (WebException ex)
            {
                lstbxFtpState.Items.Add("上传发生错误，返回信息为：" +
                lstbxFtpState.TopIndex = lstbxFtpState.Items.Count
                MessageBox.Show(ex.Message, "上传失败");
            }
        }

        private string GetUriString(string filename)
        {
            string uri = string.Empty;
            if (currentDir.EndsWith("/"))
            {
                uri = ftpUristring + currentDir + filename;
            }
            else
            {
                uri = ftpUristring + currentDir + "/" + filename;
            }

            return uri;
        }

        // 从服务器上下载文件到本地事件
        private void btndownload_Click(object sender, EventArgs e)
        {
            string fileName = GetSelectedFile();
            if (fileName.Length == 0)
            {
                MessageBox.Show("请选择要下载的文件！","提示");
                return;
            }

            // 选择保存文件的位置
            SaveFileDialog saveFileDialog = new SaveFileDialog();
            saveFileDialog.FileName = fileName;
            saveFileDialog.Filter = "所有文件(*.*)|(*.*)";
            if (saveFileDialog.ShowDialog() != DialogResult.OK)
            {
                return;
            }

            string filePath = saveFileDialog.FileName;
            try
            {
                string uri = GetUriString(fileName);
                FtpWebRequest request = CreateFtpWebRequest(uri, We
```

```
                FtpWebResponse response = GetFtpResponse(request);
                if (response == null)
                {
                    lstbxFtpState.Items.Add("服务器未响应...");
                    lstbxFtpState.TopIndex = lstbxFtpState.Items.Co
                    return;
                }

                Stream responseStream = response.GetResponseStream(
                FileStream filestream = File.Create(filePath);
                int buflength = 8196;
                byte[] buffer = new byte[buflength];
                int bytesRead =1;
                lstbxFtpState.Items.Add("打开下载通道，文件下载中...");
                while (bytesRead != 0)
                {
                    bytesRead = responseStream.Read(buffer, 0, bufl
                    filestream.Write(buffer, 0, bytesRead);
                }

                responseStream.Close();
                filestream.Close();
                lstbxFtpState.Items.Add("下载完毕，服务器返回：" + res
                lstbxFtpState.TopIndex = lstbxFtpState.Items.Count
                MessageBox.Show("下载完成！");
            }
            catch (WebException ex)
            {
                lstbxFtpState.Items.Add("发生错误，返回状态为：" + ex.
                lstbxFtpState.TopIndex = lstbxFtpState.Items.Count
                MessageBox.Show(ex.Message, "下载失败");
            }
        }

        // 获得选择的文件
        // 如果选择的是目录或者是返回上层目录，则返回null
        private string GetSelectedFile()
        {
            string filename = string.Empty;
            if (!(lstbxFtpResources.SelectedIndex == -1 || lstbxFtp
            {
                string[] namefield = lstbxFtpResources.SelectedIter
                filename = namefield[0];
            }
            return filename;

        }
        // 删除服务器文件事件
        private void btnDelete_Click(object sender, EventArgs e)
        {
            string filename = GetSelectedFile();
            if (filename.Length == 0)
            {
```

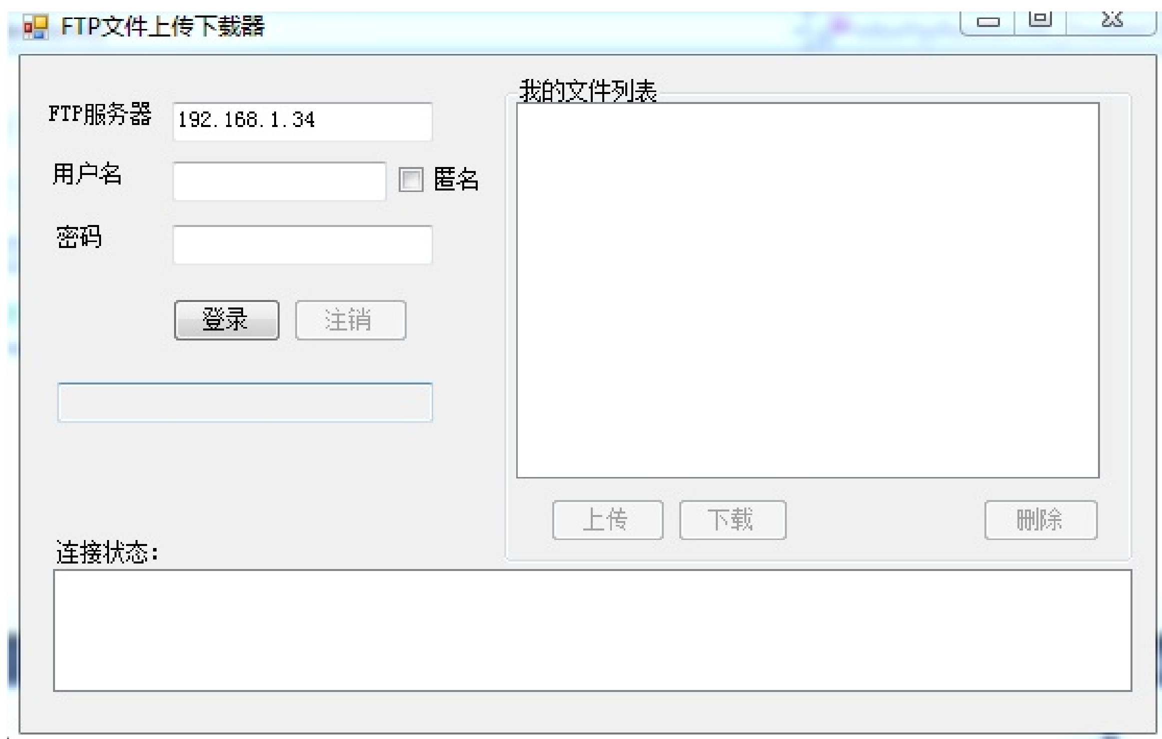
```
                MessageBox.Show("请选择要删除的文件！", "提示");
                return;
            }

            try
            {
                string uri = GetUriString(filename);
                if (MessageBox.Show("确定要删除文件 " + filename + " 
                {
                    FtpWebRequest request = CreateFtpWebRequest(uri
                    FtpWebResponse response = GetFtpResponse(reques
                    if (response == null)
                    {
                        lstbxFtpState.Items.Add("服务器未响应...");
                        lstbxFtpState.TopIndex = lstbxFtpState.Iten
                        return;
                    }

                    lstbxFtpState.Items.Add("文件删除成功，服务器返回：
                    ShowFtpFileAndDirectory();
                }
                else
                {
                    return;
                }
            }
            catch (WebException ex)
            {

                lstbxFtpState.Items.Add("发生错误，返回状态为：" + ex.
                lstbxFtpState.TopIndex = lstbxFtpState.Items.Count
                MessageBox.Show(ex.Message, "删除失败");
            }
        }
        #endregion
```

由于程序的演示效果需要结合下一专题介绍的FTP服务器，具体的演示效果大家可以查看——专题十二：实现一个简单的FTP服务器，下面就列出程序的主界面截图：



四、小结

这个专题的介绍就到这里的，在下一个专题将和大家介绍下如何实现一个FTP服务器，这样再加上本专题制作的FTP文件上传下载器就可以形成一个完整的软件套件，自己实现FTP文件上传下载器访问自己实现的FTP服务器将会让大家觉得很很有趣的， 想赶快体验下这样的一种乐趣吗？那就赶快下载本专题的源码来亲身体验下吧。通过希望通过本专题让大家对FTP协议不再陌生，并且做Asp.net开发的朋友，文件的上传和下载是一个公共模块的，然后Asp.net中的文件上传和下载只是通过浏览器向HTTP服务器发送HTTP命令，来告诉HTTP服务器说“我想和你对话”，“我想要你上面的某某文件”以及“我想上传一个文件到你的上面去”等等的对话，这个系列完成之后，我也会和大家总结下网络编程的知识的。

最后提供下源码下载地址：http://files.cnblogs.com/zhili/FTPUpDownloader.zip

[C# 网络编程系列]专题十二：实现一个简单的FTP服务器

引言：

休息一个国庆节后好久没有更新文章了，主要是刚开始休息完心态还没有调整过来的， 现在差不多进入状态了， 所以继续和大家分享下网络编程的知识，在本专题中将和大家分享如何自己实现一个简单的FTP服务器。在我们平时的上网过程中，一般都是使用FTP的客户端来对商家提供的服务器进行访问（上传、下载文件），例如我们经常用到微软的SkyDrive网盘,115网盘等，然而我们经常用到的都是网页版本的，网页版本和客户端版本的不同，网页版本的FTP客户端，它与服务器的交流是使用HTTP协议发出对服务器的请求的，而客户端版本采用的是FTP协议发出命令对服务器进行请求。然后我们接触到FTP服务器却很少的， 所以本专题中将和大家介绍下如何实现一个FTP服务器（不要觉得服务器很深奥一样的，大家可以简单的认为服务器也是一个程序，该程序是对客户端发来的请求做处理的，请求大家可以简单理解为字符串，从这个角度看， 服务器程序就是一个对字符串解析的过程。），也是为后面的一个专题做一个铺垫，因为后面专题将和大家介绍下FTP客户端——文件上传下载器，有了自己自定义的FTP服务器后， 自定义的FTP客户端就可以对自定义的FTP服务器进行访问，使两者形成一个完整的软件，从而也让大家对基于FTP协议的工具有一个初步的了解。

一、基于**FTP**协议的客户端和服务器是如何**"**沟通的**"**？

FTP客户端和FTP服务器之间的“沟通”分为四个阶段的：

1. 启动FTP

客户通过FTP客户端软件，发起FTP交互式的命令，就是告诉服务器（也就是一台电脑，服务器上与一个程序（FTP服务）会接收命令，并解析发来的命令，然后发出回复信息）说：“我想和你聊聊天，可以吗？”

2. 建立控制连接

客户端TCP层根据客户给出的服务器IP地址，向服务器提供FTP服务的21号端口发出主动建立连接的请求，服务器接收到请求后，通过3次握手之后，客户端和服务器之间就建立一个TCP连接（就是一条通道，就好比生活中马路，有了马路之后车才能够在两地之间运送东西），之后，所有用户发出的FTP命令和服务器的回应都是通过该连接来传送的， 所以也把这个TCP连接叫做控制连接，控制连接在用户退出之前一直存在。

3. 建立数据连接和进行文件传输

现在客户端和服务器端已经建立聊天的通道了（控制连接），但是两者聊天过程中如果互相想赠送礼物要怎么办呢？(这里形象的把客户端和服务器端文件的传输比喻两个人通过聊天后互相赠送礼物的过程)，此时我们就需要另外一条马路（数据连接）来进行“礼物的赠送”了，具体赠送礼物的过程如下：

1. 客户端通过控制连接向服务器发送一个上传文件的命令时，会自己分配一个临

时的TCP端口号。
2. 客户端通过控制连接向服务器发送一个命令（下面将会介绍的PORT命令）来告诉服务器自己的IP地址和临时的端口号，然后再发送一条上传文件的命令（可以理解为——客户端要送礼物给服务器时，实际上不是简单的发送一个送礼物命令的，在这之前还需要发送一条自我介绍命令（就是告诉服务器自己的IP地址和端口号）来告诉服务器自己就是刚刚和它聊天的那位，这也很符合我们日常送礼物的流程的，一般大家接到礼物都要弄明白送礼物的人是谁，是不是自己认识的）
3. 服务器接收到客户端的IP地址和临时端口号后，以这个IP地址和端口号为目标，使用服务器上的20端口（该端口是用来传输数据的端口）向客户端发出主动建立连接的请求。
4. 客户端收到请求后，通过3此握手后就与服务器之间建立了另外一条TCP连接——数据连接，即用来互相送礼物的通道。
5. 客户端在自己的文件系统中选择要赠送（上传）的文件
6. 客户端将文件写入到文件传输进程中（写入网络流中）
7. 服务器端将传输来的文件在服务器端的文件系统中进行存储
8. 文件传输完成后，由服务器主动关闭该数据的连接

4 关闭FTP

当用户退出FTP时，通关客户端发送退出命令，之后控制连接被关闭，FTP服务结束。

二、从上面的沟通过程中你明白了什么？

从上面客户端与服务器端的沟通过程中，这里可以概括几点：

（1）客户端与服务器端进行交互过程中，传输层使用的是TCP协议而不是其他传输层协议

（2）沟通过程有两条TCP连接——一条是控制连接，即传输命令和响应信息的通道，另一条是数据连接，即传输文件的马路，并且必须先有控制连接才能建立数据连接，因为要进行文件传输首先必须知道客户的IP地址和端口号，这个过程就是通过控制连接传送的命令来告知服务器客户端的IP地址和端口号，之后再在两者之间建立数据连接来传输文件

（3）在服务器端，控制连接（端口号为21）和数据连接（端口号 为20）使用了不同的端口号

三、赠送礼物的方式？——文件传输模式

客户端与FTP服务器建立数据连接之后，首先需要告诉服务器采用哪种文件传输模式，FTP提供了两种文件传输模式，一种是主动（Port）模式，另一种是被动（Passive）模式。

主动模式——服务器向客户端发起数据连接请求，被动模式——客户端向服务器发起数据请求。

然而两种模式有什么相同点和不同点呢？

两种模式的相同点：服务器都使用21号端口进行用户验证和管理

不同点：传送文件数据的方式不一样，主动模式的FTP服务器数据端口固定在20，而被动模式的FTP服务器数据端口则在1025~65535之间的随机数。

3.1 主动模式

主动模式——服务器主动连接客户端，然后传输文件，在这种模式下，FTP客户端先用一个端口N（N>1024）向服务器的21号端口发起控制连接，连接成功后，在发出PORT N+1命令告诉服务器自己监听的端口为N+1;服务器接受到该命令后，用一个新的数据端口（20号端口）与客户端的端口N+1建立连接，然后进行文件传输，而客户端则通过监听N+1端口接受文件数据。

注意：采用主动模式存在一个问题，如果客户端安装了防火墙或在内网时，由于防火墙一般不允许接受外部发起的标准端口以外的连接请求，因此外部*FTP*服务器就无法使用主动模式穿过防火墙主动连接客户端（这里与客户端连接的端口为*N+1(N>1024)*,非标准端口），从而造成无法传送文件数据，此时就需要采用被动模式传送文件了。

3.2 被动模式

被动模式——服务器被动接受客户端连接请求，即控制连接请求和数据连接请求都是由客户端发起，在这种模式下，FTP客户端先随机开始一个端口N向服务器的21号端口发起控制连接，然后向服务器发送PASV命令。服务器收到该命令后，会用一个新的端口P(P>1024)进行监听，同时将该端口号告诉客户端，客户端接受到响应命令后，再通过新的端口N+1连接服务器的端口P，然后进行文件数据传输。

注意：采用被动模式与主动模式也存在相同的问题，如果服务器安装了防火墙，客户端同样可能无法与服务器端的端口*P*建立数据请求，因为该请求可能会被防火墙过滤掉。在实际应用中，服务器一般指定一个端口范围，允许客户端与该范围内的端口建立数据连接，而不再这个范围内的端口会被服务器的防火墙过滤掉，从而在一定程度上消除了针对服务器的恶意攻击。

四、 **FTP**协议中有哪些命令的？

协议简单说就是一个规范，就好比打牌一样，制定一个大家都能明白的规则，斗地主的规则被大家都认可的，但是私下我们也可以自定义规则来玩的（例如说三个只能带一个等这样的规则），同样FTP规则也是大家都认可的一个协议，我们当然也可以自定义协议。

由于.Net平台下目前还没有提供对FTP服务器端开发的类库，因此要实现一个FTP服务器端的应用程序，就必须了解FTP协议的详细内容。

4.1 FTP命令有哪些？

FTP 协议中规定了一些大家都认识的命令和组成。FTP协议中的命令都由3~4个字母组成，命令与参数之间用空格隔开，每个命令用回车换行结束。

（1）访问命令

（1）访问命令有：

USER命令——格式为：USER <username>, 指定登录的用户名，以便服务器进行身份验证。这个命令通常是控制连接后第一个发出的命令

PASS命令——格式为：PASS <password>, 指定用户密码，该命令必须跟在登录用户名命令之后。

REIN命令——格式为：REIN， 表示重新初始化用户信息，该命令终止当前USER的传输，同时终止正在传输的数据，然后重置所有参数，并打开控制连接，以便客户端再次发生USER命令。

QUIT命令——格式为：QUIT，关闭与服务器的连接

（2）模式设置命令：

PASV命令——格式为：PASV，该命令告诉FTP服务器，让FTP服务器在指定的数据端口进行监听，被动接受客户端的请求。如果未指定任何模式，FTP服务器默认使用PASV模式

PORT命令——格式为：PORT <address>,该命令告诉FTP服务器，客户端监听的端口号是address,让FTP服务器采用主动模式连接客户端。

TYPE命令——格式为： TYPE <data type>，该命令指定要传输的数据类型，有ASCII和BINARY两种类型。

MODE命令——格式为：MODE <mode>,该命令指定传输模式，S表示流，B表示块，C表示压缩。

（3）文件管理命令

CWD命令——格式为：CWD <directory>,该命令是用户可以在不同的目录或数据集下工作而不用改变登录信息，directory一般是目录名或与系统相关的文件集合。

PWD命令——格式为：PWD，该命令返回当前工作目录。

MKD命令——格式为：MKD <directory>,该命令表示在指定路径下创建新目录，directory 表示特定目录的字符串。

CDUP命令——格式为：CDUP，该命令表示回到上层目录

RMD命令——格式为：RMD <directory>,删除指定目录，directory表示特定目录的字符串。

LIST命令——格式为：LIST <name>,该命令返回指定路径下的子目录及文件列表，name 为路径。省略路径时，返回当前路径下的文件列表。

NLIST命令——格式为：NLIST <directory>，该命令返回指定路径下的目录列表，省略路径时，返回当前目录。

RNFR命令——格式为：RNFR <old path>,该命令表示重新命名文件，该命令的下一条命令用RNTO指定新的文件名。

RNTO命令——格式为：RNTO <new path>,该命令和RNFR命令共同完成对文件的重命名。

DELE命令——格式为：DELE <filename>,该命令表示删除指定路径下的文件

（4）文件传输命令：

RETR命令——RETR <filename>,表示下载指定路径的文件

STOR命令——STOR <filename>,表示上传一个指定的文件，并将其存储在指定的位置，如果文件已存在，原文件将被覆盖，如果文件不存在，则创建新文件。

（5）其他命令

SYST命令——格式为：SYST，该命令返回服务器使用的操作系统。

4.2 FTP响应码

客户端发送FTP命令后，服务器需要返回FTP响应码，响应码即是回答，我们平常聊天中别人问了说了话或者问了问题，另外一方就需要回答，FTP协议中定义以响应码的形式来作为回答，FTP响应码由ASCII编码的3位数字开头，后面接一行文本提示信息，数字和提示信息中有一个空格，如XXX 接收请求。

每个响应码同样以回车换行结束。

FTP响应码的3位数字每位都有特定的意义，具体见下表：

	响应码	表示
第1位数字	1XX	表示信息已被服务器正确接收，但尚未被处理
	2XX	表示信息已被服务器正确处理完毕
	3XX	彪西信息已被服务器正在接受，并正在处理中
	4XX	表示信息处理错误（暂时）
	5XX	表示信息处理错误（永久）
第2位数字	X0X	表示语法错误
	X1X	表示系统状态与信息
	X2X	表示与FTP服务器系统连接状态
	X3X	表示与用户认证有关的信息
	X4X	表示未定义
	X5X	表示与文件系统有关的信息

下表列出了常用的响应码所代表的意义：

响应码	意义	响应码	意义
110	重新启动标记应答	332	登陆是需要账户信息
120	服务在指定时间内准备好	350	请求的文件操作需要进一步命令
125	数据连接打开——开始传输	421	服务关闭
150	文件状态良好，将要打开数据连接	425	不能打开数据连接
200	命令成功	426	关闭连接，终止传输
202	命令没有执行	450	文件不可用
211	系统状态回复	451	中止请求操作:有本地错误
212	目录状态回复	452	磁盘空间不足
213	文件状态回复	500	无效命令
214	帮助信息回复	501	语法错误
215	系统类型回复	502	命令未执行
220	服务就绪	503	命令顺序错误
221	服务关闭控制连接，可以退出登陆	504	无效命令参数
225	数据连接打开，无传输正在进行	530	未登陆
226	关闭数据连接，请求的文件操作成功	532	存储文件需要账户信息
227	进入被动模式	550	未执行请求操作
230	用户已登陆	551	请求操作终止:页类型未知
250	请求的文件操作完成	552	请求文件操作终止：超过存储分配
257	创建路径名	553	为执行请求的操作:文件名不合法
331	用户名正确，需要口令		

五、实现自定义的**FTP**服务器

相信大家看完上面的介绍对FTP协议以及FTP客户端和FTP服务器的交互过程有一定的理解的，这时候大家知道理论后就一定很想知道知道这些之后可以做什么的？答案就是可以制作一个简单的FTP服务器，大家可以根据代码来进一步理解FTP协

议。下面是程序中一些核心代码片段：

View Code

```
// 启动服务器
        private void btnFtpServerStartStop_Click(object sender, Eve
        {
            if (myTcpListener == null)
            {
                listenThread = new Thread(ListenClientConnect);
                listenThread.IsBackground = true;
                listenThread.Start();

                lstboxStatus.Enabled = true;
                lstboxStatus.Items.Clear();
                lstboxStatus.Items.Add("启动Ftp服务...");
                btnFtpServerStartStop.Text = "停止";
            }
            else
            {
                myTcpListener.Stop();
                myTcpListener = null;
                listenThread.Abort();
                lstboxStatus.Items.Add("Ftp服务已停止！");
                lstboxStatus.TopIndex = lstboxStatus.Items.Count -

                btnFtpServerStartStop.Text = "启动";
            }
        }

        // 监听端口，处理客户端连接
        private void ListenClientConnect()
        {
            myTcpListener = new TcpListener(IPAddress.Parse(tbxFtpS
            // 开始监听传入的请求
            myTcpListener.Start();
            AddInfo("启动成功！");
            AddInfo("Ftp服务运行中...[单机”停止“退出]");
            while (true)
            {
                try
                {
                    // 接收连接请求
                    TcpClient tcpClient = myTcpListener.AcceptTcpCl
                    AddInfo(string.Format("客户端（{0}）与本机（{1}）建
                    User user = new User();
                    user.commandSession = new UserSeesion(tcpClient
                    user.workDir = tbxFtpRoot.Text;
                    Thread t = new Thread(UserProcessing);
                    t.IsBackground = true;
                    t.Start(user);
                }
```

```
                catch
                {
                    break;
                }
            }
        }

        // 处理客户端用户请求
        private void UserProcessing(object obj)
        {
            User user = (User)obj;
            string sendString = "220 FTP Server v1.0";
            RepleyCommandToUser(user, sendString);
            while (true)
            {
                string receiveString = null;
                try
                {
                    // 读取客户端发来的请求信息
                    receiveString = user.commandSession.streamReade
                }
                catch(Exception ex)
                {
                    if (user.commandSession.tcpClient.Connected ==
                    {
                        AddInfo(string.Format("客户端({0}断开连接！)"
                    }
                    else
                    {
                        AddInfo("接收命令失败！" + ex.Message);
                    }

                    break;
                }

                if (receiveString == null)
                {
                    AddInfo("接收字符串为null,结束线程！");
                    break;
                }

                AddInfo(string.Format("来自{0}：[{1}]", user.commanc

                // 分解客户端发来的控制信息中的命令和参数
                string command = receiveString;
                string param = string.Empty;
                int index = receiveString.IndexOf(' ');
                if (index != -1)
                {
                    command = receiveString.Substring(0, index).ToU
                    param = receiveString.Substring(command.Length)
                }
```

```
                    // 处理不需登录即可响应的命令（这里只处理QUIT）
                    if (command == "QUIT")
                    {
                        // 关闭TCP连接并释放与其关联的所有资源
                        user.commandSession.Close();
                        return;
                    }
                    else
                    {
                        switch (user.loginOK)
                        {
                            // 等待用户输入用户名：
                            case 0:
                                CommandUser(user, command, param);
                                break;

                            // 等待用户输入密码
                            case 1:
                                CommandPassword(user, command, param);
                                break;

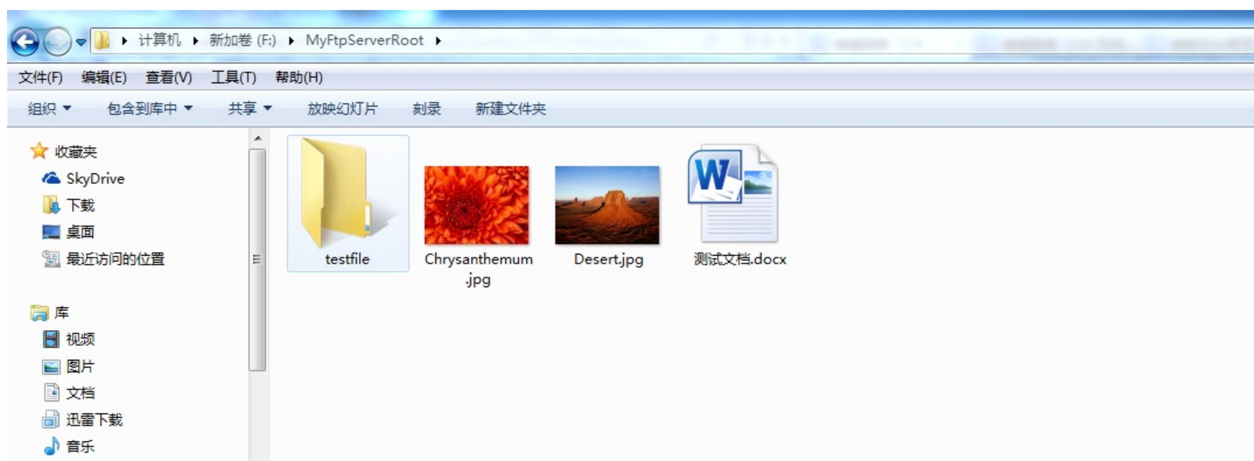
                            // 用户名和密码验证正确后登陆
                            case 2:
                                switch (command)
                                {
                                    case "CWD":
                                        CommandCWD(user, param);
                                        break;
                                    case "PWD":
                                        CommandPWD(user);
                                        break;
                                    case "PASV":
                                        CommandPASV(user);
                                        break;
                                    case "PORT":
                                        CommandPORT(user, param);
                                        break;
                                    case "LIST":
                                        CommandLIST(user, param);
                                        break;
                                    case "NLIST":
                                        CommandLIST(user, param);
                                        break;
                                    // 处理下载文件命令
                                    case "RETR":
                                        CommandRETR(user, param);
                                        break;
                                    // 处理上传文件命令
                                    case "STOR":
                                        CommandSTOR(user, param);
                                        break;
                                    // 处理删除命令
                                    case "DELE":
```

```
                            CommandDELE(user, param);
                            break;
                        // 使用Type命令在ASCII和二进制模式进行变
                        case "TYPE":
                            CommandTYPE(user, param);
                            break;
                        default:
                            sendString = "502 command is no
                            RepleyCommandToUser(user, sendS
                            break;
                    }

                    break;
                }
            }
        }
    }
```

程序演示截图：

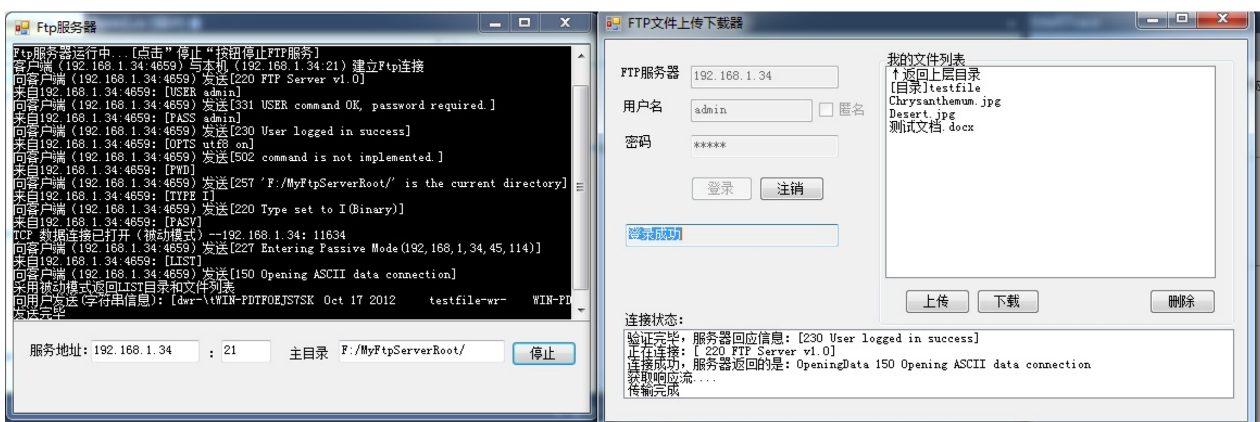
首先在F:\盘下新建文件夹MyFtpServerRoot,在其中创建目录结构并放一些文件资源，例如图片，文档等，程序中演示的目录结构如下图：



这样，本地的FTP服务站点就已经建好了，运行FTP服务器程序，然后点击“启动”按钮后就启动了FTP服务器，运行结果如下图所示：



然后配合上个专题中实现的FTP客户端来完成与FTP服务器的“聊天”演示，因为FTP服务器程序中已经初始化用户名和密码（都为admin），所以FTP客户端中取消选择“匿名复选框”，直接输入用户名和密码为admin后点击“登录”按钮后就完成了用户验证的过程，并与FTP服务器建立了控制连接和数据连接。运行结果如下图：



当然用户可以通过"上传"、“下载”和删除按钮来对FTP服务器上的文件进行操作，这里就不贴出运行图片了，大家可以下载源码来测试下的。

六、内容的结尾，说说后面的计划吧

这个专题介绍完后，我这个C#网络编程系列也就介绍完了，这个系列中主要介绍网络编程的一些入门知识，对于朋友在留言中经常提到的“打洞”技术以及一些网络编程中一些更难的内容还大家一起努力来学习的，同时我也会在后面和大家分享下一些实际开发过程中的网络编程的内容（在后面的文章打算和大家分享一个下载器的实现），最后，希望这个系列可以让大家对网络协议有一个最初的入门，这样在实际的开发过程中才知道这些实现背后的原理。之后我总结下我这个系列的所有文章的索引，以便让大家更好的阅读和查找关于这个系列的所有文章。

源码下载：http://files.cnblogs.com/zhili/FtpServer.zip，大家如果觉得不错的话，还请大家推荐下，谢谢大家的支持

用来演示的服务器目录：http://files.cnblogs.com/zhili/MyFtpServerRoot.zip

上个专题FTP文件上传下载器源码：http://files.cnblogs.com/zhili/FTPUpDownloader.zip

C# 互操作性入门系列

C# 互操作性入门系列(一)：C#中互操作性介绍

C#互操作系列文章：

本专题概要：

- 引言
- 平台调用
- **C++ Interop(互操作)**
- **COM Interop(互操作)**

一、引言

这个系列是在C#基础知识中遗留下来的一个系列的，因为在C# 4.0中的一个新特性就是对COM互操作改进，然而COM互操作性却是.NET平台下其中一种互操作技术，为了帮助大家更好的了解.NET平台下的互操作技术，所以才有了这个系列。然而有些朋友们可能会有这样的疑问——“为什么我们需要掌握互操作技术的呢？”对于这个问题的解释就是——掌握了.NET平台下的互操作性技术可以帮助我们在.NET中调用非托管的dll和COM组件。.NET是建立在操作系统的之上的一个开发框架，其中.NET 类库中的类也是对Windows API的抽象封装，然而.NET类库不可能对所有Windows API进行封装，当.NET中没有实现某个功能的类，然而该功能在Windows API被实现了，此时我们完全没必要去自己在.NET中自定义个类，这时候就可以调用Windows API 中的函数来实现，此时就涉及到托管代码与非托管代码的交互，此时就需要使用到互操作性的技术来实现托管代码和非托管代码更好的交互。.NET 平台下提供了3种互操作性的技术：

1. Platform Invoke(P/Invoke)，即平台调用,主要用于调用C库函数和Windows API
2. C++ Introp, 主要用于Managed C++(托管C++)中调用C++类库
3. COM Interop, 主要用于在.NET中调用COM组件和在COM中使用.NET程序集。

下面就对这3种技术分别介绍下。

二、平台调用

使用平台调用的技术可以在托管代码中调用动态链接库（Dll）中实现的非托管函数，如Win32 Dll和C/C++ 创建的dll。看到这里，有些朋友们应该会有疑问——在怎样的场合我们可以使用平台调用技术来调用动态链接库中的非托管函数呢？

这个问题就如前面引言中说讲到的一样，当在开发过程中，.NET类库中没有提供相关API然而Win32 API 中提供了相关的函数实现时，此时就可以考虑使用平台调用的技术在.NET开发的应用程序中调用Win32 API中的函数；

然而还有一个使用场景就是——由于托管代码的效率不如非托管代码，为了提高效率，此时也可以考虑托管代码中调用C库函数。

2.1 在托管代码中通过平台调用来调用非托管代码的步骤

(1). 获得非托管函数的信息，即dll的名称，需要调用的非托管函数名等信息

(2). 在托管代码中对非托管函数进行声明，并且附加平台调用所需要属性

(3). 在托管代码中直接调用第二步中声明的托管函数

2.2 平台调用的调用过程

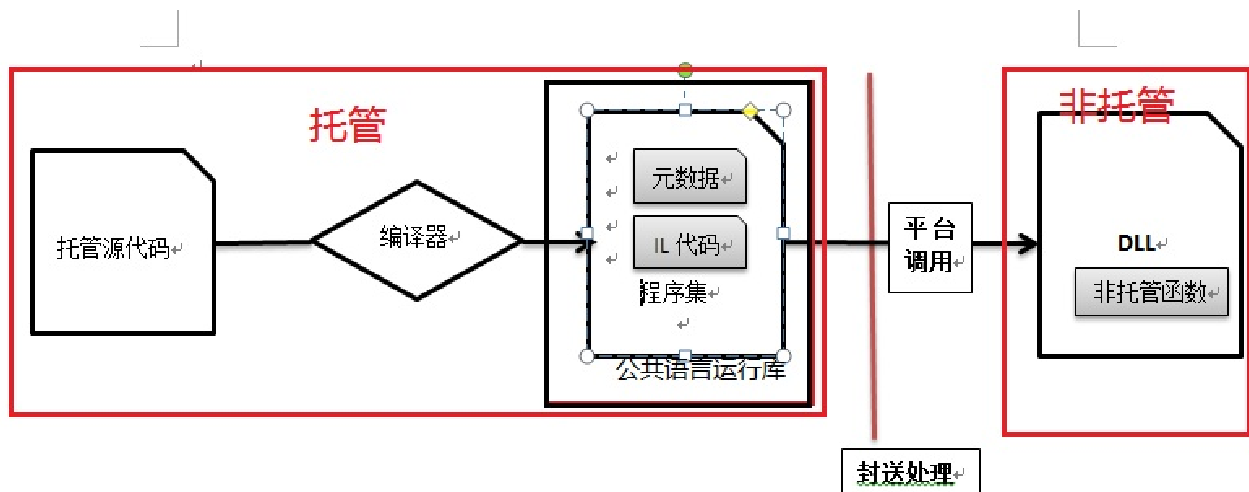
(1) 查找包含该函数的DLL，当需要调用某个函数时，当然第一步就需要知道包含该函数的DLL的位置，所以平台调用的第一步也就是查找DLL，其实在托管代码中调用非托管代码的调用过程可以想象成叫某个人做事情，首先我们要找到那个人在哪里（即查找函数的**DLL**过程），找到那个人之后需要把要做的事情告诉他（相当于加载**DLL**到内存中和传入参数），最后让他去完成需要完成的事情（相当于让非托管函数去执行任务）。

(2) 将找到的DLL加载到内存中。

(3) 查找函数在内存中的地址并把其参数推入堆栈，来封送所需的数据。CLR只会在第一次调用函数时，才会去查找和加载DLL，并查找函数在内存中的地址。当函数被调用过一次之后，CLR会将函数的地址缓存起来，CLR这种机制可以提高平台调用的效率。在应用程序域被卸载之前，找到的DLL都一直存在于内存中。

(4) 执行非托管函数。

平台调用的过程可以通过下图更好地理解：



三、**C++ Interop**

第二部分主要向大家介绍了第一种互操作性技术，然后我们也可以使用C++ Interop技术来实现与非托管代码进行交互。然而C++ Interop 方式有一个与平台调用不一样的地方，就是C++ Interop 允许托管代码和非托管代码存在于一个程序集中，甚至同一个文件中。C++ Interop 是在源代码上直接链接和编译非托管代码来实现与非托管代码进行互操作的，而平台调用是加载编译后生成的非托管DLL并查找函数

的入口地址来实现与非托管函数进行互操作的。**C++ Interop**使用托管**C++**来包装非托管**C++**代码，然后编译生成程序集，然后再托管代码中引用该程序集，从而来实现与非托管代码的互操作。 关于具体的使用和与平台调用的比较，这里就不多介绍，我将会在后面的专题中具体介绍。

四、**COM Interop**

COM（Component Object Model,组件对象模型）是微软之前推荐的一个开发技术，由于微软过去十多年里面开发了大量的COM组件，然而不可能在使用.NET技术重写这些COM组件实现的功能，所以为了解决在.NET中的托管代码能够调用COM组件的问题，.NET 平台下提供了COM Interop,即COM互操作技术，COM Interop不仅支持在托管代码中使用COM组件，而且还支持想CMO组件功能托管对象。下面就这两种支持分别做一个介绍。

4.1 在**.NET**中使用**COM**组件

在.NET中使用COM对象，主要有3种方法：

1.
    i.
        i. 使用TlbImp工具为COM组件创建一个互操作程序集来绑定早期的COM对象，这样就可以在程序中添加互操作程序集来调用COM对象
        ii. 通过反射来后期绑定COM对象
        iii. 通过P/Invoke创建COM对象或使用C++ Interop为COM对象编写包装类

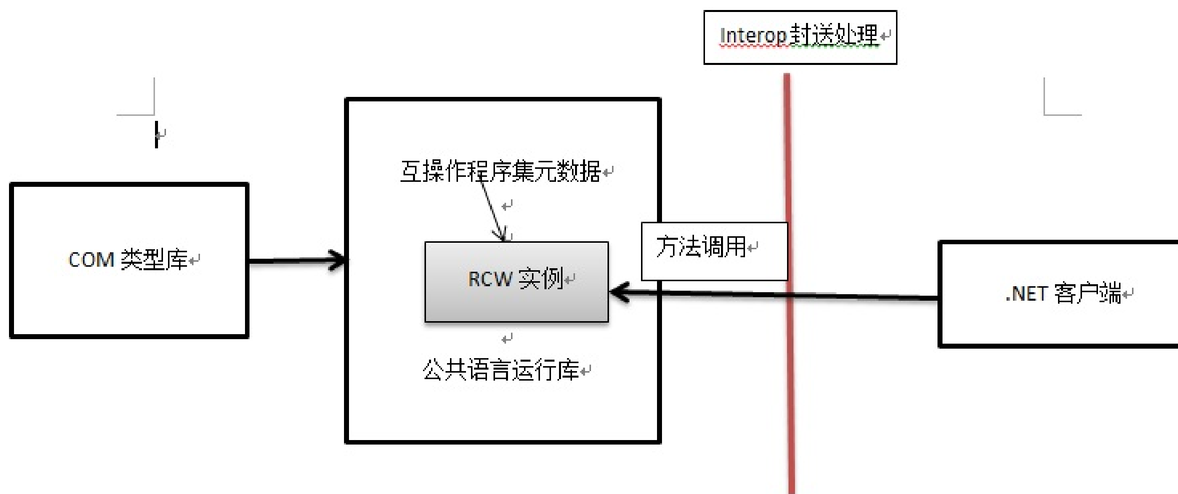
但是我们经常使用的都是方法一，下面介绍下使用方法一在.NET 中使用COM对象的步骤：

1. 找到要使用的COM 组件并注册它。使用 regsvr32.exe 注册或注销 COM DLL。
2. 在项目中添加对 COM 组件或类型库的引用。

Tlbimp.exe (Type Library Importer), which takes a type library as input, to output a .NET Framework interop assembly." data-guid="e48b737e46c03f731589e3d6eae713c7">添加引用时，**Visual Studio** 会用到**Tlbimp.exe**（类型库导入程序），**Tlbimp.exe**程序将生成一个 **.NET Framework** 互操作程序集。**该程序集又称为运行时可调用包装 (RCW)，其中包含了包装COM组件中的类和接口。Visual Studio 将生成组件的引用添加至项目。**

3. 创建RCW中类的实例，这样就可以使用托管对象一样来使用COM对象。

下面通过一个图更好地说明在.NET中使用COM组件的过程：

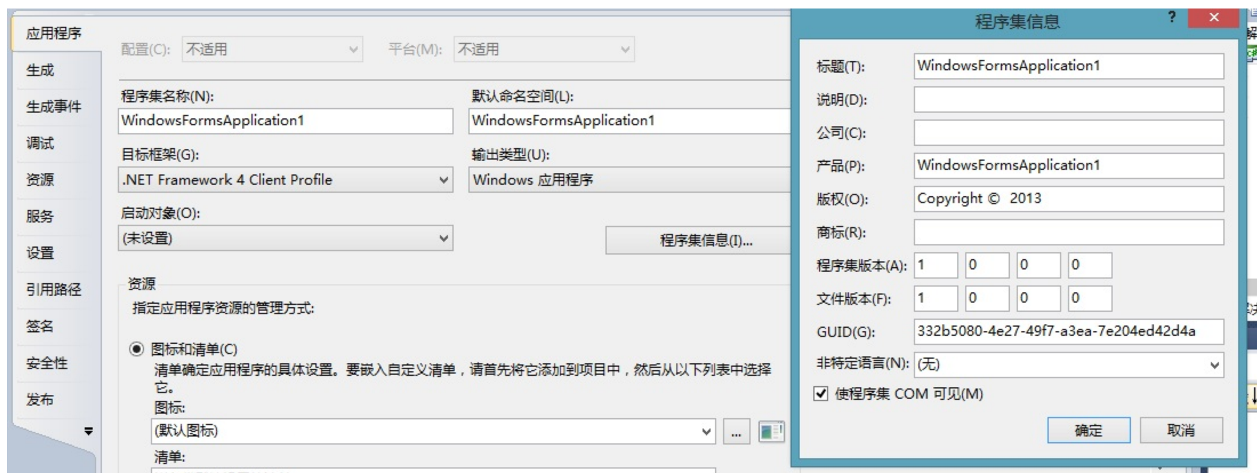


4.2 在COM中使用.NET程序集

.NET 公共语言运行时通过COM可调用包装（COM Callable Wrapper,即CCW）来完成与COM类型库的交互。CCW可以使COM客户端认为是在与普通的COM类型交互，同时使.NET组件认为它正在与托管应用程序交互。在这里CCW是非托管COM客户端与托管对象之间的一个代理。 CCW既可以维护托管对象的生命周期，也负责数据类型在COM和.NET之间的相互转换。实现在COM使用.NET 类型的基本步骤如：

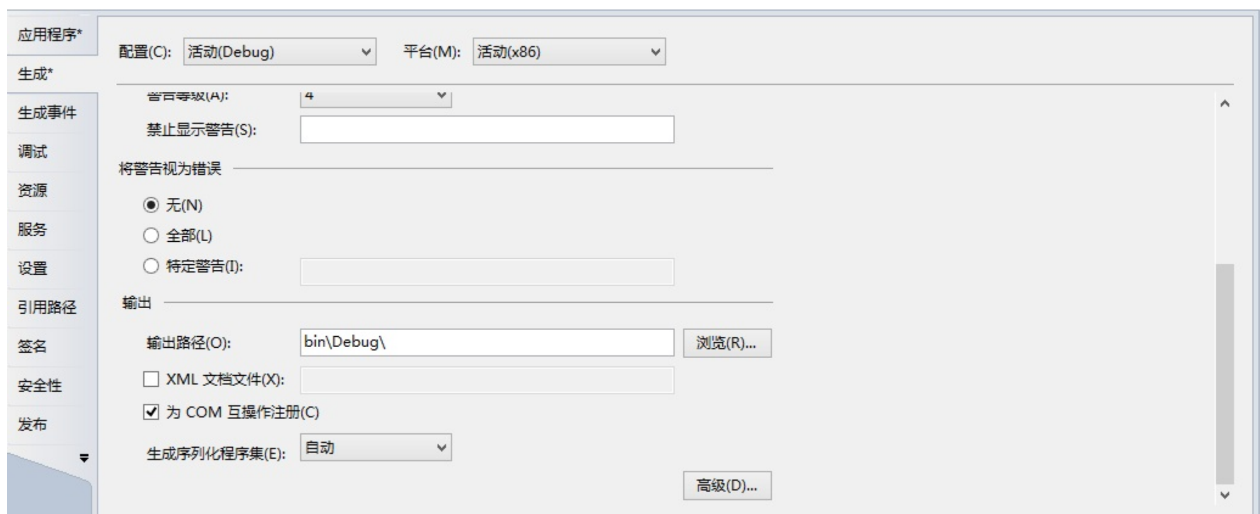
1. 在C#项目中添加互操作特性

可以修改C#项目属性使程序集对COM可见。右键解决方案选择属性，在“应用程序标签”中选择“程序集信息”按钮，在弹出的对话框中选择 “使程序集COM可见” 选项，如下图所示：



2. 生成COM类型库并对它进行注册以供COM客户端使用

在“生成”标签中，选中 “为COM互操作注册”选项，如下图：



勾选“为COM互操作注册”选项后，Visual Studio会调用类型库导出工具(Tlbexp.exe)为.NET程序集生成COM类型库再使用程序集注册工具(Regasm.exe)来完成对.NET程序集和生成的COM类型库进行注册，这样COM客户端可以使用CCW服务来对.NET对象进行调用了。

五、总结

介绍到这里，本专题的内容就结束，本专题主要对.NET 提供的互操作的技术做了一个总的概括，在后面的专题中将会对具体的技术进行详细的介绍和给出一些简单的使用例子。

C# 互操作性入门系列(二)：使用平台调用调用 Win32 函数

C#互操作系列文章：

1. **C#**互操作性入门系列**(**一**)**：**C#**中互操作性介绍
2. **C#**互操作性入门系列**(**二**)**：使用平台调用调用**Win32** 函数
3. **C#**互操作性入门系列**(**三**)**：平台调用中的数据封送处理
4. **C#**互操作性入门系列**(**四**)**：在**C#** 中调用**COM**组件

本专题概要**:**

- 引言
- 如何使用平台调用**Win32** 函数**——**从实例开始
- 当调用**Win32**函数出错时怎么办？**——**获得**Win32**函数的错误信息
- 小结

一、引言

上一专题对.NET 互操作性做了一个全面的概括，其中讲到.NET平台下实现互操作性有三种技术——平台调用，C++ Interop和COM Interop,今天在这个专题中将会大家介绍第一种技术,即平台调用。然而朋友们应该会有这样的疑问，平台调用到底有什么用呢？ 为什么我们要用平台调用的技术了？对于这两个问题的答案就是**——**平台调用可以帮助我们实现在**.NET**平台下**(**也就是指用**C#**、**VB.net**语言写的应用程序下**)**可以调用非托管函数（指定的是**C/C++**语言写的函数）。这样如果我们在.NET平台下实现的功能有现有的C/C++ 函数实现了这样的功能，这时候我们完全没必要自己再用托管语言（如C#、vb.net）去实现一个这样的功能，这时候我们应该想到"拿来主义"，直接使用平台调用技术调用C/C++ 实现的函数。然而在实际应用中，使用平台调用技术来调用Win32 API较为普遍，所以在这个专题中将为大家具体介绍了如何使用平台调用来调用Win32函数以及调用过程中应该注意的问题，下面就从一个具体的实例开始本专题的介绍。

二、如何使用平台调用**Win32** 函数**——**从实例开始

在前一个专题中已经介绍了使用平台调用来调用非托管函数的步骤:

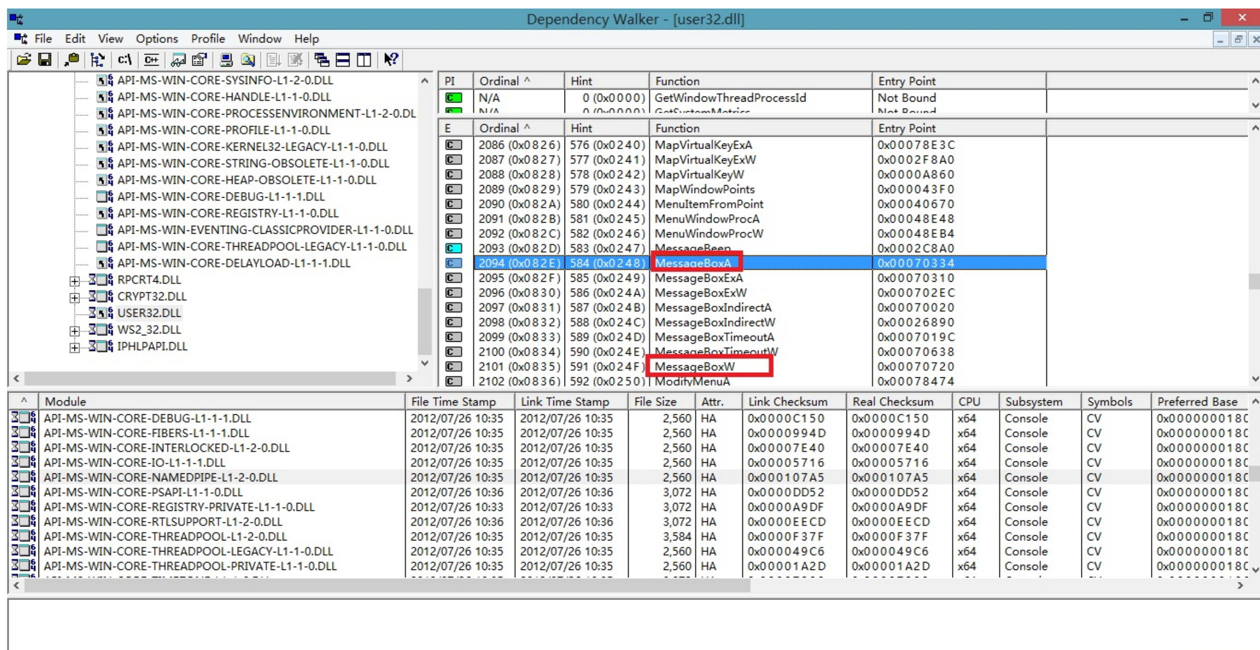
(1). 获得非托管函数的信息，即**dll**的名称，需要调用的非托管函数名等信息

(2). 在托管代码中对非托管函数进行声明，并且附加平台调用所需要属性

(3). 在托管代码中直接调用第二步中声明的托管函数

然而调用Win32 API函数还有一些问题需要注意的地方, 首先, 因为很多**Win32 API** 函数都有**ANSI**和**Unicode**两个版本，所以在托管代码声明时需要指定调用调用函数的版本。 然而很多Win32 API函数有ANSI和Unicode两个版本并不是随便说说的，而是有根据的。大家从调用步骤中可以看出，第一步就需要知道非托管函数声明，为了找到需要需要调用的非托管函数，可以借助两个工具——Visual Studio自带的**dumpbin.exe**和**depends.exe**，dumpbin.exe 是一个命令行工具，可以用于查看从

非托管DLL中导出的函数等信息，可以通过打开Visual Studio 2010 Command Prompt(中文版为Visual Studio 命令提示(2010))，然后切换到DLL所在的目录，输入 dummbin.exe/exports dllName, 如 **dummbin.exe/exports User32.dll** 来查看User32.dll中的函数声明，关于更多命令的参数可以参看MSDN；然而depends.exe是一个可视化界面工具，大家可以从 "**VS安装目录\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\Tools\Bin**" 这个路径找到，然后双击 depends.exe 就可以出来一个可视化界面（如果某些人安装的VS没有附带这个工具，也可以从官方网站下载：http://www.dependencywalker.com/），如下图：



上图中 我用红色标示出 MessageBox 有两个版本，而MessageBoxA 代表的就是ANSI版本，而MessageBoxW 代笔的就是Unicode版本，这也是上面所说的依据。下面就看看 MessageBox的C++声明的(更多的函数的定义大家可以从MSDN中找到，这里提供MessageBox的定义在MSDN中的链接：http://msdn.microsoft.com/en-us/library/windows/desktop/ms645505(v=vs.85).aspx.aspx))：

现在已经知道了需要调用的Win32 API 函数的定义声明，下面就依据平台调用的步骤，在.NET 中实现对该非托管函数的调用，下面就看看.NET中的代码的：

```
using System;

**// 使用平台调用技术进行互操作性之前，首先需要添加这个命名空间** using Syst

namespace 平台调用Demo
{
    class Program
    {
        // 在托管代码中对非托管函数进行声明，并且附加平台调用所需要属性
        // 在默认情况下，CharSet为CharSet.Ansi
        // 指定调用哪个版本的方法有两种——通过DllImport属性的CharSet字段和i
      **  // 在托管函数中声明注意一定要加上 static 和extern 这两个关键字**
        [DllImport("user32.dll")]
```

```
        public static extern int MessageBox1(IntPtr hWnd, String te

        // 在默认情况下，CharSet为CharSet.Ansi
        [DllImport("user32.dll")]
        public static extern int MessageBoxA(IntPtr hWnd, String te

        // 在默认情况下，CharSet为CharSet.Ansi
        [DllImport("user32.dll")]
        public static extern int MessageBox(IntPtr hWnd, String tex

        // 第一种指定方式，通过CharSet字段指定
        [DllImport("user32.dll", CharSet = CharSet.Unicode)]
        public static extern int MessageBox2(IntPtr hWnd, String te

        // 通过EntryPoint字段指定
        [DllImport("user32.dll", EntryPoint="MessageBoxA")]
        public static extern int MessageBox3(IntPtr hWnd, String te

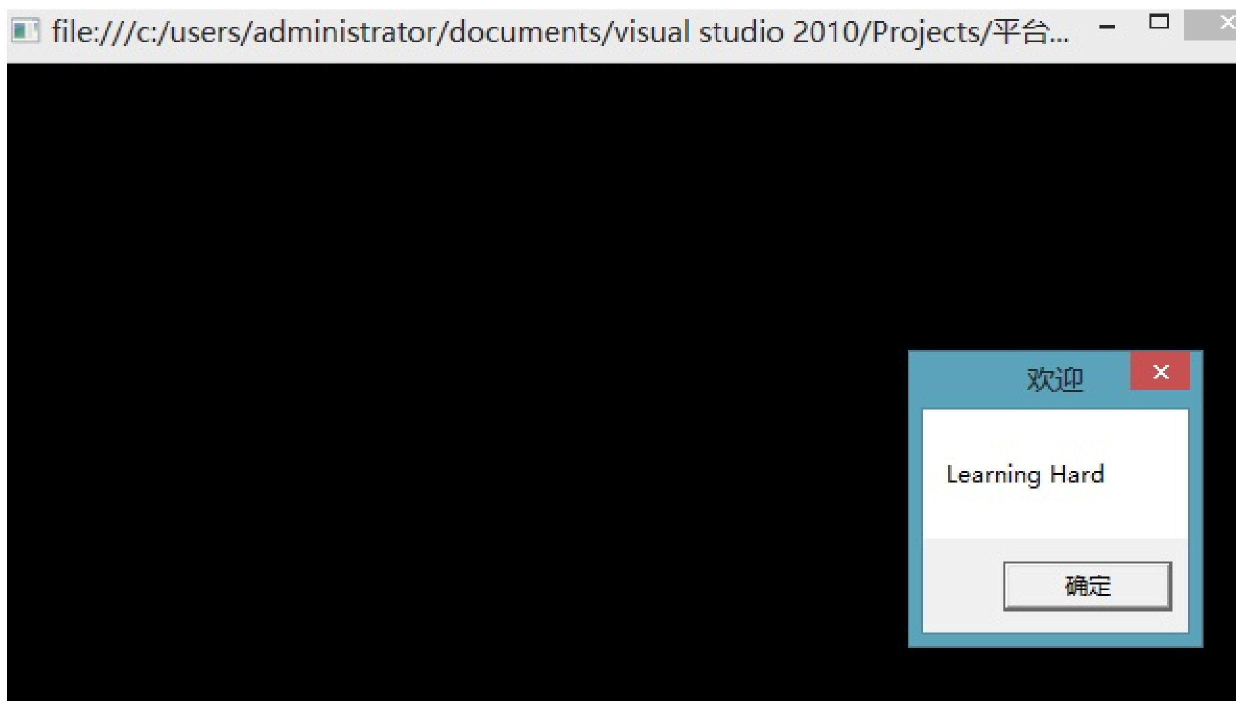
        [DllImport("user32.dll", EntryPoint = "MessageBoxW")]
        public static extern int MessageBox4(IntPtr hWnd, String te
        static void Main(string[] args)
        {
            // 在托管代码中直接调用声明的托管函数
            // 使用CharSet字段指定的方式，要求在托管代码中声明的函数名必须与
            // 否则就会出现找不到入口点的运行时错误
            //MessageBox1(new IntPtr(0), "Learning Hard", "欢迎", 0

            // 下面的调用都可以运行正确
            //MessageBoxA(new IntPtr(0), "Learning Hard", "欢迎", 0
            //MessageBox(new IntPtr(0), "Learning Hard", "欢迎", 0)

            // 使用指定函数入口点的方式调用
            //MessageBox3(new IntPtr(0), "Learning Hard", "欢迎", 0

            // 调用Unicode版本的会出现乱码
            MessageBox4(new IntPtr(0), "Learning Hard", "欢迎", 0);
        }
    }
}
```

运行正确的结果为：



从代码的注释中可以看出，第一个调用MessageBox1会出现运行时错误，然而为什么改调用会出现 “无法在 DLL“user32.dll”中找到名为“MessageBox1”的入口点。”的错误呢？ 为了知道为什么，这里就需要明白通过CharSet字段指定的这种方式的内部执行行为了。之所以会出现这个错误，是因为当指定CharSet为Ansi时，P/Invoke首先会通过根函数名在User32.dll中搜索，即不带后缀A的函数名MessageBox1 进行搜索，如果找到与跟函数一样名称的函数，就调用该函数；

如果没有找到则使用带后缀为A的函数MessageBox1A进行搜索，如果找到，则使用该函数，如果还是没有找到，则会出现 “无法在 DLL“user32.dll”中找到名为“MessageBox1”的入口点。”的错误。把CharSet指定为Unicode时，搜索方式是一样的，只是没找到根函数时会加W后缀进行搜索的。 从上面的搜索调用函数的过程中可以发现，因为user32.dll中既不存在MessageBox1函数也不存在MessageBox1A函数，所以才会出现调用错误。(朋友看到出现错误时，应该会有这样的疑问——我们如何捕捉错误来显示错误信息呢？这个疑问将会在下一部分解释。) 然而使用平台调用技术中，还需要注意下面4点：

(1). DllImport属性的**ExactSpelling**字段如果设置为**true**时，则在托管代码中声明的函数名必须与要调用的非托管函数名完全一致,因为从ExactSpelling.aspx)字面意思可以看出为 "准确拼写"的意思,当**ExactSpelling**设置为**true**时，此时会改变平台调用的行为，此时平台调用只会根据根函数名进行搜索,而找不到的时候不会添加 A或者W来进行再搜索,. MessageBox, platform invoke searches for **MessageBox** and fails when it cannot locate the exact spelling." data-guid="f890f75ea0bb3ec1eaa56ed400e56d45">例如，如果指定 **MessageBox**，则平台调用将搜索 **MessageBox**，如果它找不到完全相同的拼写则会出现找不到入口函数的错误。 从前面的代码中可以看出，我们在代码中并没有指定 **ExactSpelling** 字段，然而代码中却没有出现调用错误，这就说明在C#和托管C++语言中，**ExactSpelling** 默认值就是false的，然而在VB。NET中，**ExactSpelling**的默认值就是true, 所以以上代码如果转化为Vb.NET时，就需要显式指定**ExactSpelling** 字段为false,不然就会出现 “找不到函数入口的错误”。 为了让大家更加容易理解上面的理论,相信大家看到下面一张图会更加理解 **ExactSpelling**字段的含义的：

MessageBox, platform invoke searches for **MessageBox** and fails when it cannot locate the exact spelling." data-guid="f890f75ea0bb3ec1eaa56ed400e56d45">

```
// 第一种指定方式，通过CharSet字段指定
// 下面显式把ExactSpelling 设置为True时，此时需要声明的函数名必须与调用的函数名完全一致才能调用成功
// 否则会出现错误
[DllImport("user32.dll", CharSet = CharSet.Unicode, ExactSpelling = true)]
public static extern int MessageBox2(IntPtr hWnd, String text, String caption, uint type);
static void Main(string[] args)
{
    MessageBox2(new IntPtr(0), "Learning Hard", "欢迎", 0);

    // 在托管代码中直接调用声明的托管函数
    // 使用CharSet字段指定的方式，要求在托管代码中声明的函
    // 否则就会出现找不到入口点的运行时错误
```

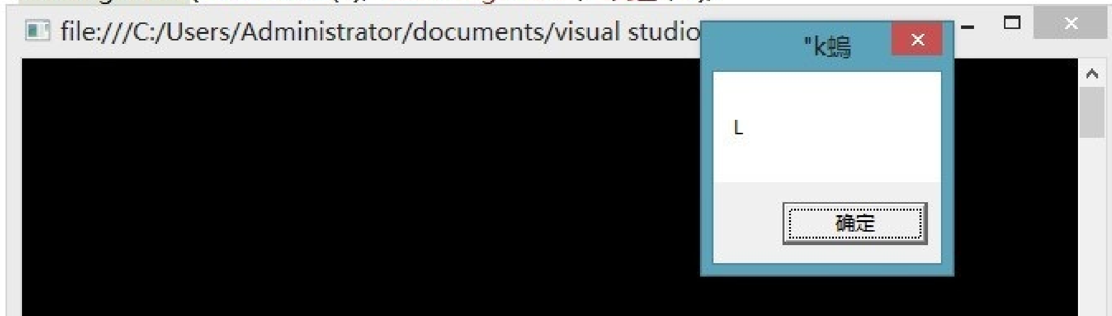


MessageBox, platform invoke searches for **MessageBox** and fails when it cannot locate the exact spelling." data-guid="f890f75ea0bb3ec1eaa56ed400e56d45">
(2). 如果采用设置CharSet的值来控制调用函数的版本时，则需要在托管代码中声明的函数名必须与根函数名一致,否则也会调用出错,这点从平台调用过程中可以很好地理解,如果需要调用非托管函数名为 MessageBoxA,而你在托管代码声明为MessageBox1,这样在搜索过程中明显就会提示找不到函数名的错误, 也就是上面代码中第一个调用出错的原因。

MessageBox, platform invoke searches for **MessageBox** and fails when it cannot locate the exact spelling." data-guid="f890f75ea0bb3ec1eaa56ed400e56d45">
(3). 如果通过指定DllImport属性的**EntryPoint**字段的方式来调用函数版本时，此时必须相应地指定与之匹配的CharSet设置，意思就是——如果指定EntryPoint为MessageBoxW,那么必须将CharSet指定为CharSet.Unicode,如果指定EntryPoint为MessageBoxA,那么必须将CharSet指定为CharSet.Ansi或者不指定，因为 CharSet默认值就是Ansi。上面代码**MessageBox4**的调用之所以会出现乱码，是因为**CharSet**指定为**Ansi(**也是默认值**)**时**,** 平台调用将字符串按照**ANSI**编码方式封送到非托管内存中**(**在**.NET** 中，字符串的编码方式默认为**Unicode**的**)**，即每个字符仅占一个字节，**(**而对于**Unicode**编码的字符串来说，字符串中的每个字符都是使用两个字节进行编码的**)**，当非托管函数**MessageBoxW**开始执行时，它会把该内存中的数据按照**Unicode**编码处理，即每两个字节当做是一个**Unicode**字符，知道遇到双字节的**'\0'** 字符结束。所以非托管函数返回的结果也就出现乱码了。 如果指定**EntryPoint** 字段的值为**MessageBoxA,**却把**CharSet**字段设置为**CharSet.Unicode**的情况下，也会出现同样的乱码问题**,**如下图所示**:**

MessageBox, platform invoke searches for **MessageBox** and fails when it cannot locate the exact spelling." data-guid="f890f75ea0bb3ec1eaa56ed400e56d45">

```
[DllImport("user32.dll", EntryPoint = "MessageBoxA", CharSet = CharSet.Unicode)]
public static extern int MessageBox5(IntPtr hWnd, String text, String caption, uint type);
static void Main(string[] args)
{
    MessageBox5(new IntPtr(0), "Learning Hard", "欢迎", 0);
```



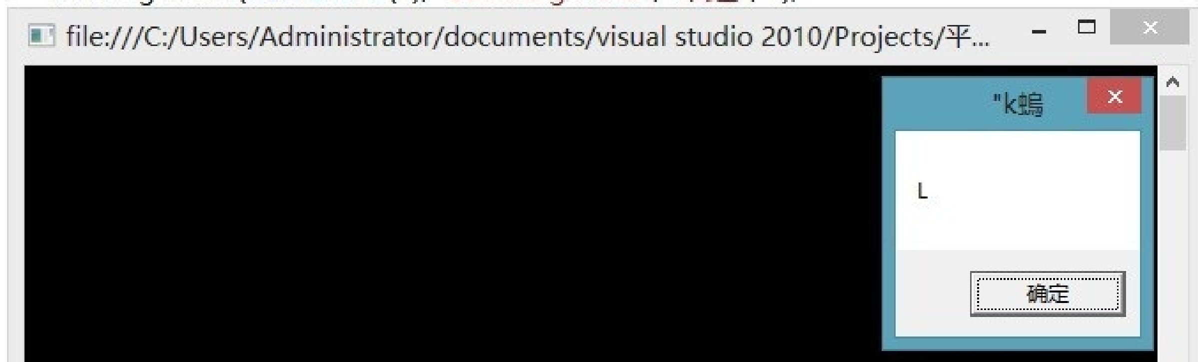
MessageBox, platform invoke searches for **MessageBox** and fails when it cannot locate the exact spelling." data-guid="f890f75ea0bb3ec1eaa56ed400e56d45">
(4). CharSet还有一个可选字段为——CharSet.Auto, 如果把CharSet字段设置为CharSet.Auto，则平台调用会针对目标操作系统适当地自动封送字符串。Unicode on Windows NT, Windows 2000, Windows XP, and the Windows Server 2003 family; the default is Ansi on Windows 98 and Windows Me." data-guid="275999a6d1aff7e55c2abc3094f769ed">在 Windows NT、Windows 2000、Windows XP 和 Windows Server 2003 系列上，默认值为 Unicode；在 Windows 98 和 Windows Me 上，默认值为 Ansi。Auto, languages may override this default." data-guid="9676c285887e8c44ff8277aed31aadd1">尽管公共语言运行时默认值为 Auto，但使用语言可重写此默认值。Ansi." data-guid="0cb899b48496e2934a217215e86dfe6a">例如，默认情况下，C# 将所有方法和类型都标记为 Ansi。所以下面的调用一样也会出现乱码,原因在第三点中已经解释了,下面直接附上测试例子和结果:

```
class Program
    {
        [DllImport("user32.dll", EntryPoint = "MessageBoxA", CharSe
        public static extern int MessageBox5(IntPtr hWnd, String te
        static void Main(string[] args)
        {
            MessageBox5(new IntPtr(0), "Learning Hard", "欢迎", 0);
         }
     }
```

MessageBox, platform invoke searches for **MessageBox** and fails when it cannot locate the exact spelling." data-guid="f890f75ea0bb3ec1eaa56ed400e56d45">Ansi." data-guid="0cb899b48496e2934a217215e86dfe6a">运行结果为：

MessageBox, platform invoke searches for **MessageBox** and fails when it cannot locate the exact spelling." data-guid="f890f75ea0bb3ec1eaa56ed400e56d45">Ansi." data-guid="0cb899b48496e2934a217215e86dfe6a">

```
[DllImport("user32.dll", EntryPoint = "MessageBoxA", CharSet = CharSet.Auto)]
public static extern int MessageBox5(IntPtr hWnd, String text, String caption, uint type);
static void Main(string[] args)
{
    MessageBox5(new IntPtr(0), "Learning Hard", "欢迎", 0);
```



三、当调用**Win32**函数出错时怎么办？——获得**Win32**函数的错误信息

前面部分为大家演示了平台调用的使用以及使用过程需要注意的问题, 当大家了解了这些之后,肯定会有这样的一个疑问,当调用Win32函数过程中遇到由Win32函数返回的错误要怎样去处理呢？ 或者由非托管函数的托管定义导致的错误或异常怎么捕捉，就如上面代码中调用**MessageBox1**出现异常时，如何捕捉并给用于一个友好的提示信息呢？对于这个两个问题,下面通过两个具体的例子来演示。

捕捉由托管定义导致的异常演示代码：

```
class Program
    {
        // 在托管代码中对非托管函数进行声明，并且附加平台调用所需要属性
        // 在默认情况下，CharSet为CharSet.Ansi
        // 指定调用哪个版本的方法有两种——通过DllImport属性的CharSet字段和通
        [DllImport("user32.dll")]
        public static extern int MessageBox1(IntPtr hWnd, String te
        static void Main(string[] args)
        {
            try
            {
                MessageBox1(new IntPtr(0), "Learning Hard", "欢迎",
            }
            catch (DllNotFoundException dllNotFoundExc)
            {
                Console.WriteLine("DllNotFoundException 异常发生，异常
            }
            catch (EntryPointNotFoundException entryPointExc)
            {
                Console.WriteLine("EntryPointNotFoundException 异常
            }
            Console.Read();
       }
}
```

运行结果为：



捕获由Win32函数本身返回异常的演示代码如下：

```
using System;
using System.ComponentModel;
// 使用平台调用技术进行互操作性之前，首先需要添加这个命名空间
using System.Runtime.InteropServices;

namespace 处理Win32函数返回的错误
{
    class Program
    {
        // Win32 API
        //  DWORD WINAPI GetFileAttributes(
        //  _In_  LPCTSTR lpFileName
        //);

        // 在托管代码中对非托管函数进行声明
        [DllImport("Kernel32.dll",**SetLastError**=true,CharSet=Cha
        public static extern uint GetFileAttributes(string filename

        static void Main(string[] args)
        {
            // 试图获得一个不存在文件的属性
            // 此时调用Win32函数会发生错误
            GetFileAttributes("FileNotexist.txt");

            // 在应用程序的Bin目录下存在一个test.txt文件,此时调用会成功
            //GetFileAttributes("test.txt");

            // 获得最后一次获得的错误
            int lastErrorCode = Marshal.GetLastWin32Error();

            // 将Win32的错误码转换为托管异常
            //Win32Exception win32exception = new Win32Exception();
            Win32Exception win32exception = new Win32Exception(last
            if (lastErrorCode != 0)
            {
                Console.WriteLine("调用Win32函数发生错误，错误信息为 :
            }
            else
            {
                Console.WriteLine("调用Win32函数成功,返回的信息为： {0}
            }

            Console.Read();
        }
    }
}
```

运行结果为:



要想获得在调用Win32函数过程中出现的错误信息，首先必须将DllImport属性的SetLastError字段设置为true,只有这样，平台调用才会将最后一次调用Win32产生的错误码保存起来，然后会在托管代码调用Win32失败后，通过Marshal类的静态方法GetLastWin32Error获得由平台调用保存的错误码，从而对错误进行相应的分析和处理。这样就可以获得Win32中的错误信息了。

上面代码简单地演示了如何在托管代码中获得最后一次发生的Win32错误信息,然而还可以通过调用Win32 API 提供的**FormatMessage**函数的方式来获得错误信息，然而这种方式有一个很显然的弊端(所以这里就不演示了)，当对**FormatMessage**函数调用失败时，这时候就有可能获得不正确的错误信息，所以，推荐采用.NET提供的Win32Exception异常类来获得具体的错误信息。关于更多的**FormatMessage**函数可以参考MSDN: http://msdn.microsoft.com/en-us/library/ms679351(v=vs.85).aspx.aspx)

四、小结

讲到这里，本专题的内容也就介绍完了，本专题只是简单介绍了使用平台调用技术来调用Win32函数，然而实际的操作远远不是这么简单的，要掌握平台调用的技术，还需要大家在工作过程多多实践。因为在本专题中涉及了一些数据封送一些知识，为了帮助大家更好掌握数据封送处理，在一个专题将为大家带来平台调用中的数据封送处理专题。

C# 互操作性入门系列(三)：平台调用中的数据封送处理

C#互操作系列文章：

本专题概要

- 数据封送介绍
- 封送Win32数据类型
- 封送字符串的处理
- 封送结构体的处理
- 封送类的处理
- 小结

一、数据封送介绍

看到这个专题时，大家的第一个疑问肯定是——什么是数据封送呢？(这系列专题中采用假设朋友的提问方式来解说概念，就是希望大家带着问题去学习本专题内容，以及大家在平时的学习过程中也可以采用这个方式，个人觉得这个方式可以使自己学习效率有所提高，即使这样在学习的过程可能会显得慢了，但是这种方式会对你所看过的知识点会有一个更深的印象。远比看的很快，最后却发现记住的没多少强，在这里分享下这个学习方式，认为可以接受的朋友可以在平时的学习中可以尝试下的，如果觉得不好的话，相信大家肯定也会有自己更好的学习方式的。)对于这个问题的解释是，数据封送是——在托管代码中对非托管函数进行互操作时，需要通过方法的参数和返回值在托管内存和非托管内存之间传递数据的过程，数据封送处理的过程是由**CLR(**公共语言运行时**)**的封送处理服务**(**即封送拆送器**)**完成的。

封送拆送器主要进行3项任务：

1. 将数据从托管类型转换为非托管类型，或从非托管类型转换为托管类型
2. 将经过类型转换的数据从托管代码内存复制到非托管内存，或从非托管内存复制到托管内存
3. 调用完成后，释放封送处理过程中分配的内存

二、封送**Win32**数据类型

对非托管代码进行互操作时，一定会有数据的封送处理。然而封送时需要处理的数据类型分为两种——可直接复制到本机结构中的类型(blittable)和非直接复制到本机结构中的类型(non-bittable)。下面就这两种数据类型分别做一个介绍。

2.1 可直接复制到本机结构中的类型

由于在托管代码和非托管代码中，数据类型在托管内存和非托管内存的表示形式不一样，因为这样的原因，所以我们需要对数据进行封送处理，以至于在托管代码中调用非托管函数时，把正确的传入参数传递给非托管函数和把正确的返回值返回给托管代码中。然而，并不是所有数据类型在两者内存的表现形式不一样的，这时候我们把在托管内存和非托管内存中有相同表现形式的数据类型称为——可直接复制到本机结构中的类型，这些数据类型不需要封送拆送器进行任何特殊的处理就可以在托管和非托管代码之间传递，下面列出一些课直接复制到本机结构中的简单数据类型：

Windows 数据类型	非托管数据类型	托管数据类型	托管数据类型解释
BYTE/Uchar/UInt8	unsigned char	**System.Byte**	无符号8位整型
Sbyte/Char/Int8	char	**System.SByte**	有符号8位整型
Short/Int16	short	**System.Int16**	有符号1位整型
USHORT/WORD/UInt16/WCHAR	unsigned short	**System.UInt16**	无符号1位整型

Bool/HResult/Int/Long	long/int	**System.Int32**	有符号3位整型
DWORD/ULONG/UINT	unsigned long/unsigned int	**System.UInt32**	无符号3位整型
INT64/LONGLONG	_int64	**System.Int64**	有符号6位整型
UINT64/DWORDLONG/ULONGLONG	_uint64	**System.UInt64**	无符号6位整型
INT_PTR/hANDLE/wPARAM	void*/int或_int64	**System.IntPtr**	有符号指针类型
HANDLE	void*	**System.UIntPtr**	无符号指针类型
			单精

FLOAT	float	**System.Single**	度浮点数
DOUBLE	double	**System.Double**	双精度浮点数

除了上表列出来的简单类型之外，还有一些复制类型也属于可直接复制到本机结构中的数据类型：

（1） 数据元素都是可直接复制到本机结构中的一元数组，如整数数组，浮点数组等

（2）只包含可直接复制到本机结构中的格式化值类型

（3）成员变量全部都是可复制到本机结构中的类型且作为格式化类型封送的类

上面提到的格式化指的是——在类型定义时，成员的内存布局在声明时就明确指定的类型。在代码中用StructLayout属性修饰被指定的类型，并将StructLayout的LayoutKind属性设置为Sequential或Explicit，例如：

```
using System.Runtime.InteropServices;

// 下面的结构体也属于可直接复制到本机结构中的类型
[StructLayout(LayoutKind.Sequential)]
public struct Point {
   public int x;
   public int y;
}
```

2.2 非直接复制到本机结构中的类型

如果一个类型不是可直接复制到本机结构中的类型，那么它就是非直接复制到本机结构中的类型。由于一些类型在托管内存和非托管内存的表现形式不一样，所以对于这种类型，封送器需要对它们进行相应的类型转换之后再复制到被调用的函数中，下面列出一些非直接复制到本机结构中的数据类型：

Windows 数据类型	非托管数据类型
Bool	bool
WCHAR/TCHAR	char/ wchar_t
LPCSTR/LPCWSTR/LPCTSTR/LPSTR/LPWSTR/LPTSTR	const char/*const wchar_t*/char/*wcha*
LPSTR/LPWSTR/LPTSTR	Char/*wchar_t*

除了上表中列出的类型之外，还有很多其他类型属于非直接复制到本机结构中的类型，例如其他指针类型和句柄类型等。理解了blittable和non-blittable类型的区别之后，就可以在互操作过程更好地处理数据的封送，下面就具体的一些数据类型的封送问题做一个简单介绍

三、封送字符串的处理

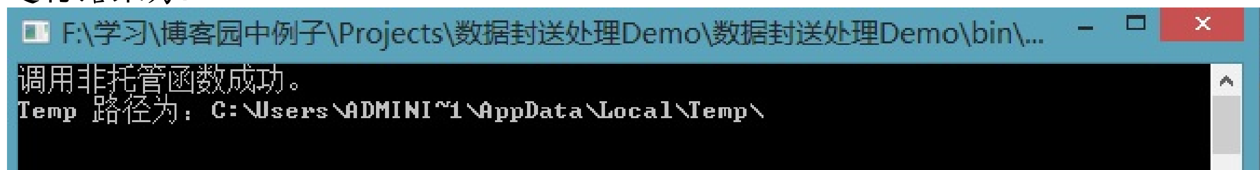
在上一个专题中，我们已经涉及到字符串的封送问题了(上一个专题中使用了将字符串作为In参数传递给Win32 MessageBox 函数，具体可以查看上一个专题) 。所以在这部分将介绍——封送作为返回值的字符串,下面是一段演示代码,代码中主要是调用Win32 GetTempPath函数来获得返回返回临时路径,此时拆送器就需要把返回的字符串封送回托管代码中。

```
// 托管函数中的返回值封送回托管函数的例子
    class Program
    {

        // Win32 GetTempPath函数的定义如下：
//DWORD WINAPI GetTempPath(
//   _In_   DWORD nBufferLength,
//   _Out_  LPTSTR lpBuffer
//);    **// 主要是注意如何在托管代码中定义该函数原型 **
        [DllImport("Kernel32.dll", CharSet = CharSet.Unicode, SetLa
        public static extern uint GetTempPath(int bufferLength, Str
        static void Main(string[] args)
        {
            StringBuilder buffer = new StringBuilder(300);
            uint tempPath=GetTempPath(300, buffer);
            string path = buffer.ToString();
            if (tempPath == 0)
            {
                int errorcode =Marshal.GetLastWin32Error();
                Win32Exception win32expection = new Win32Exception(
                Console.WriteLine("调用非托管函数发生异常，异常信息为：" 
            }

            Console.WriteLine("调用非托管函数成功。");
            Console.WriteLine("Temp 路径为：" + buffer);
            Console.Read();
        }
    }
```

运行结果为:



四、封送结构体的处理

在我们实际调用Win32 API函数时，经常需要封送结构体和类等复制类型，下面就以Win32 函数GetVersionEx为例子来演示如何对作为参数的结构体进行封送处理。为了在托管代码中调用非托管代码,首先我们就要知道非托管函数的定义,下面是GetVersionEx非托管定义(更多关于该函数的信息可以参看MSDN链接:http://msdn.microsoft.com/en-us/library/ms885648.aspx)：

参数lpVersionInformation是一个指向 **OSVERSIONINFO**结构体的指针类型,所以我们在托管代码中为函数GetVersionEx函数之前，必须知道 **OSVERSIONINFO**结构体的非托管定义，然后再在托管代码中定义一个等价的结构体类型作为参数。以下是**OSVERSIONINFO**结构体的非托管定义：

```
typedef struct  _OSVERSIONINFO{
    DWORD dwOSVersionInfoSize;       //在使用GetVersionEx之前要将此初始
    DWORD dwMajorVersion;               //系统主版本号
    DWORD dwMinorVersion;               //系统次版本号
    DWORD dwBuildNumber;               //系统构建号
    DWORD dwPlatformId;                  //系统支持的平台
    TCHAR szCSDVersion[128];          //系统补丁包的名称
    WORD wServicePackMajor;            //系统补丁包的主版本
    WORD wServicePackMinor;            //系统补丁包的次版本
    WORD wSuiteMask;                      //标识系统上的程序组
    BYTE wProductType;                    //标识系统类型
    BYTE wReserved;                         //保留,未使用
} OSVERSIONINFO;
```

知道了**OSVERSIONINFO**结构体在非托管代码中的定义之后, 现在我们就需要在托管代码中定义一个等价的结构，并且要保证两个结构体在内存中的布局相同。托管代码中的结构体定义如下：

```
  // 因为Win32 GetVersionEx函数参数lpVersionInformation是一个指向 OSVE
        // 所以托管代码中定义个结构体，把结构体对象作为非托管函数参数
        **[StructLayout(LayoutKind.Sequential,CharSet=CharSet.Unico
        {
            public UInt32 OSVersionInfoSize; // 结构的大小，在调用方法
            public UInt32 MajorVersion; // 系统主版本号
            public UInt32 MinorVersion; // 系统此版本号
            public UInt32 BuildNumber;  // 系统构建号
            public UInt32 PlatformId;  // 系统支持的平台

            // 此属性用于表示将其封送成内联数组
            [MarshalAs(UnmanagedType.ByValTStr,SizeConst=128)]
            public string CSDVersion; // 系统补丁包的名称
            public UInt16 ServicePackMajor; // 系统补丁包的主版本
            public UInt16 ServicePackMinor;  // 系统补丁包的次版本
            public UInt16 SuiteMask;   //标识系统上的程序组
            public Byte ProductType;    //标识系统类型
            public Byte Reserved;  //保留,未使用
        }
```

从上面的定义可以看出, 托管代码中定义的结构体有以下三个方面与非托管代码中的结构体是相同的:

- 字段声明的顺序
- 字段的类型
- 字段在内存中的大小

并且在上面结构体的定义中,我们使用到了 **StructLayout** 属性，该属性属于 System.Runtime.InteropServices命名空间(所以在使用平台调用技术必须添加这个额外的命名空间)。这个类的作用就是允许开发人员显式指定结构体或类中数据字段的内存布局,为了保证结构体中的数据字段在内存中的顺序与定义时一致,所以指定为 **LayoutKind.Sequential**（该枚举也是默认值)。 下面就具体看看在托管代码中调用的代码：

```
using System;
using System.ComponentModel;
using System.Runtime.InteropServices;
namespace 封送结构体的处理
{
    class Program
    {
        // 对GetVersionEx进行托管定义
        **// 为了传递指向结构体的指针并将初始化的信息传递给非托管代码，需要用
        // 这里不能使用out关键字，如果使用了out关键字，CLR就不会对参数进行初
        [DllImport("Kernel32",CharSet=CharSet.Unicode,EntryPoint="G
        private static extern Boolean GetVersionEx_Struct(ref  OSVe

        // 因为Win32 GetVersionEx函数参数lpVersionInformation是一个指向
        // 所以托管代码中定义个结构体，把结构体对象作为非托管函数参数
        [StructLayout(LayoutKind.Sequential,CharSet=CharSet.Unicode
        public struct OSVersionInfo
        {
            public UInt32 OSVersionInfoSize; // 结构的大小，在调用方法
            public UInt32 MajorVersion; // 系统主版本号
            public UInt32 MinorVersion; // 系统此版本号
            public UInt32 BuildNumber;  // 系统构建号
            public UInt32 PlatformId;  // 系统支持的平台

            // 此属性用于表示将其封送成内联数组
            [MarshalAs(UnmanagedType.ByValTStr,SizeConst=128)]
            public string CSDVersion; // 系统补丁包的名称
            public UInt16 ServicePackMajor; // 系统补丁包的主版本
            public UInt16 ServicePackMinor;  // 系统补丁包的次版本
            public UInt16 SuiteMask;   //标识系统上的程序组
            public Byte ProductType;    //标识系统类型
            public Byte Reserved;  //保留,未使用
        }

        // 获得操作系统信息
        private static string GetOSVersion()
        {
            // 定义一个字符串存储版本信息
            string versionName = string.Empty;

            // 初始化一个结构体对象
            OSVersionInfo osVersionInformation = new OSVersionInfo(

            // 调用GetVersionEx 方法前，必须用SizeOf方法设置结构体中OSVe
            osVersionInformation.OSVersionInfoSize = (UInt32)Marsha
```

```
            // 调用Win32函数
            Boolean result = GetVersionEx_Struct(ref osVersionInfor

            if (!result)
            {
                // 如果调用失败，获得最后的错误码
                int errorcode = Marshal.GetLastWin32Error();
                Win32Exception win32Exc = new Win32Exception(error
                Console.WriteLine("调用失败的错误信息为： " + win32Exc

                // 调用失败时返回为空字符串
                return string.Empty;
            }
            else
            {
                Console.WriteLine("调用成功");
                switch (osVersionInformation.MajorVersion)
                {
                    // 这里仅仅讨论 主版本号为6的情况，其他情况是一样讨论的
                    case 6:
                        switch (osVersionInformation.MinorVersion)
                        {
                            case 0:
                                if (osVersionInformation.ProductTyp
                                {
                                    versionName = " Microsoft Windo
                                }
                                else
                                {
                                    versionName = "Microsoft Window
                                }
                                break;
                            case 1:
                                if (osVersionInformation.ProductTyp
                                {
                                    versionName = " Microsoft Windo
                                }
                                else
                                {
                                    versionName = "Microsoft Window
                                }
                                break;
                            case 2:
                                versionName = "Microsoft Windows 8'
                                break;
                        }
                        break;
                    default:
                        versionName = "未知的操作系统";
                        break;
                }
                return versionName;
```

```
            }
        }

        static void Main(string[] args)
        {
            string OS=GetOSVersion();
            Console.WriteLine("当前电脑安装的操作系统为：{0}", OS);
            Console.Read();
        }
    }
}
```

运行结果为:

```
file:///F:/学习/博客园中例子/Projects/数据封送处理Demo/封送结构体的处理/...
调用成功
当前电脑安装的操作系统为：Microsoft Windows 8
```

附上微软操作系统名和版本号的对应关系,大家可以参考下面的表对上面代码进行其他的讨论:

操作系统	版本号
Windows 8	6.2
Windows 7	6.1
Windows Server 2008 R2	6.1
Windows Server 2008	6.0
Windows Vista	6.0
Windows Server 2003 R2	5.2
Windows Server 2003	5.2
Windows XP	5.1
Windows 2000	5.0

五、封送类的处理

下面直接通过GetVersionEx函数进行封送类的处理的例子，具体代码如下：

```
using System;
using System.ComponentModel;
using System.Runtime.InteropServices;

namespace 封送类的处理
{
    class Program
```

```
    {
        // 对GetVersionEx进行托管定义
        // 由于类的定义中CSDVersion为String类型，String是非直接复制到本机
        // 所以封送拆送器需要进行复制操作。
        // 为了是非托管代码能够获得在托管代码中对象设置的初始值(指的是OSVers
        // 所以必须加上[In]属性；函数返回时，为了将结果复制到托管对象中，必须
        // 这里不能是用ref关键字，因为 OsVersionInfo是类类型，本来就是引用
        [DllImport("Kernel32", CharSet = CharSet.Unicode, EntryPoin
        private static extern Boolean GetVersionEx_Struct([In, Out]

        // 获得操作系统信息
        private static string GetOSVersion()
        {
            // 定义一个字符串存储操作系统信息
            string versionName = string.Empty;

            // 初始化一个类对象
            OSVersionInfo osVersionInformation = new OSVersionInfo(

            // 调用Win32函数
            Boolean result = GetVersionEx_Struct(osVersionInformati

            if (!result)
            {
                // 如果调用失败，获得最后的错误码
                int errorcode = Marshal.GetLastWin32Error();
                Win32Exception win32Exc = new Win32Exception(errorc
                Console.WriteLine("调用失败的错误信息为： " + win32Exc

                // 调用失败时返回为空字符串
                return string.Empty;
            }
            else
            {
                Console.WriteLine("调用成功");
                switch (osVersionInformation.MajorVersion)
                {
                    // 这里仅仅讨论 主版本号为6的情况，其他情况是一样讨论的
                    case 6:
                        switch (osVersionInformation.MinorVersion)
                        {
                            case 0:
                                if (osVersionInformation.ProductTyp
                                {
                                    versionName = " Microsoft Windo
                                }
                                else
                                {
                                    versionName = "Microsoft Window
                                }
                                break;
                            case 1:
                                if (osVersionInformation.ProductTyp
```

```
                                {
                                    versionName = " Microsoft Windo
                                }
                                else
                                {
                                    versionName = "Microsoft Window
                                }
                                break;
                            case 2:
                                versionName = "Microsoft Windows 8'
                                break;
                        }
                        break;
                    default:
                        versionName = "未知的操作系统";
                        break;
                }
                return versionName;
            }
        }

        static void Main(string[] args)
        {
            string OS = GetOSVersion();
            Console.WriteLine("当前电脑安装的操作系统为：{0}", OS);
            Console.Read();
        }
    }

    [StructLayout(LayoutKind.Sequential, CharSet = CharSet.Unicode)
    public class OSVersionInfo
    {
        public UInt32 OSVersionInfoSize = (UInt32)Marshal.SizeOf(ty
        public UInt32 MajorVersion = 0;
        public UInt32 MinorVersion = 0;
        public UInt32 BuildNumber = 0;
        public UInt32 PlatformId = 0;

        // 此属性用于表示将其封送成内联数组
        [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 128)]
        public string CSDVersion = null;

        public UInt16 ServicePackMajor = 0;
        public UInt16 ServicePackMinor = 0;
        public UInt16 SuiteMask = 0;

        public Byte ProductType = 0;
        public Byte Reserved;
    }
}
```

运行结果还是和上面使用结构体定义的一样,还是附上下图吧:



六、小结

本专题主要介绍了几种类型的数据封送处理, 对于封送处理的一句话概括就是——保证托管代码中定义的数据在内存中的布局与非托管代码中的内存布局相同，专题中也列出了一些简单类型在非托管代码和托管代码中定义的对应关系，对于一些没有列出来的指针类型或回调函数等可以使用万能的**IntPtr**类型在托管代码中定义.然而本专题只是对数据封送做一个入门的介绍, 要真真掌握数据封送处理还要考虑很多其他的因素,这个就需要大家在平时工作中积累的。

C# 互操作性入门系列(四)：在C# 中调用COM组件

C#互操作系列文章：

1. **C#**互操作性入门系列(一)：**C#**中互操作性介绍
2. **C#**互操作性入门系列(二)：使用平台调用调用**Win32** 函数
3. **C#** 互操作性入门系列(三)：平台调用中的数据封送处理
4. **C#**互操作性入门系列(四)：在**C#** 中调用**COM**组件

本专题概要：

- 引言
- 如何在**C#**中调用**COM**组件——访问**Office** 互操作对象
- 在**C#** 中调用**COM**组件的实现原理剖析
- 错误处理
- 小结

一、引言

COM（Component Object Modele,组件对象模型）是微软以前推崇的一个开发技术，所以现在微软的很多产品都用到了COM组件，如Office,IE 等。然而如果.NET平台下的程序想访问COM组件的方式来实现某个功能怎么办呢？ 正是由于开发人员有这个需求，所以微软在.NET FrameWork中为COM和托管代码之间进行互操作提供了支持，这种互操作性的技术就是**COM Interop**。 但是COM Interop(COm互操作)这项技术，不仅支持在托管代码中使用COM对象，并且也支持在COM中使用托管对象，本专题只针对在.NET中调用COM对象来介绍，由于COM技术现在用的不多，所以如何在COM中使用托管对象将不会在本系列中做出介绍，如果有需要的朋友可以参看MSDN的相关链接：http://msdn.microsoft.com/zh-cn/library/3y76b69k(v=vs.100).aspx.aspx)。

下面就从一个具体的实例来看看在.NET 中是如何调用COM组件的。

二、如何在**C#**中调用**COM**组件——访问**Office** 互操作对象

因为Office产品中使用了很多COM组件，下面就演示通过调用Office中的COM对象来创建Word文档并保存创建的文档到文件目录下的例子（在新建的控制台程序里添加"**Microsoft.Office.Interop.Word 14.0.0.0** " 这个引用，14.0.0.0版本是对应于Office 2010的一个互操作程序集， 12.0.0.0版本则是对应于Office 2007的互操作程序集，如果你电脑中只安装了Office 2007的话，就只能找到 12.0.0.0的版本的，如果安装了Office 2010的话，就可以同时找到这两个版本。）。具体代码如下：

```
using System;
// 添加额外的命名空间
using Microsoft.Office.Interop.Word;

namespace COM互操作性
{
    class Program
    {
        static void Main(string[] args)
        {
            // 调用COM对象来创建Word文档
            CreateWordDocument();
        }

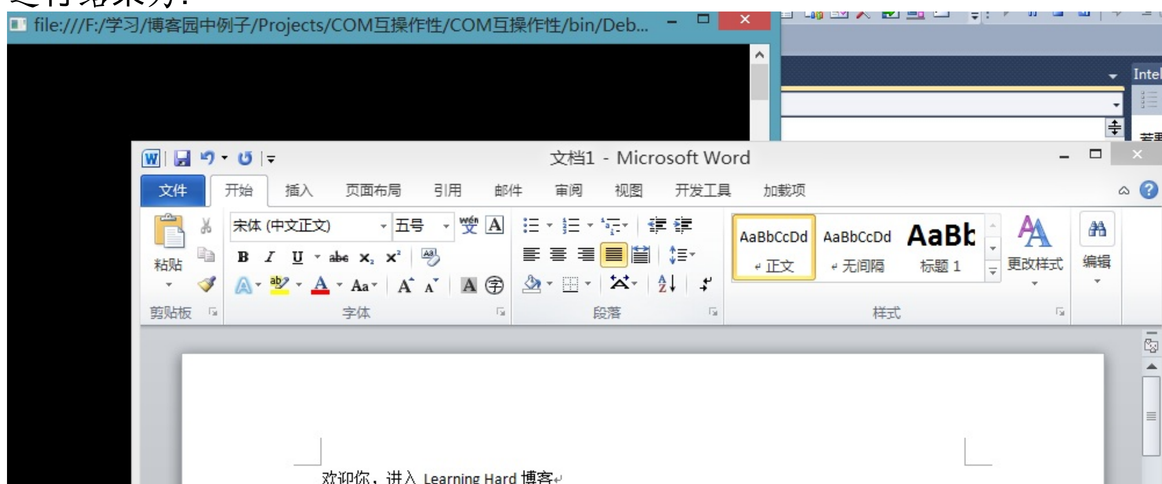
        private static void CreateWordDocument()
        {
            // 启动Word并使Word可见
            Application wordApp = new Application() { Visible = tru

            // 新建Word文档
            wordApp.Documents.Add();
            Document wordDoc = wordApp.ActiveDocument;
            Paragraph para = wordDoc.Paragraphs.Add();
            para.Range.Text = "欢迎你，进入Learning Hard博客";

            // 保存文档
            object filename = @"D:\learninghard.doc";
            wordDoc.SaveAs2(filename);

            // 关闭Word
            wordDoc.Close();
            wordApp.Application.Quit();
        }
    }
}
```

运行结果为:



此时在所指定的文件目录中就可以看到你刚才创建的Word文档了。通过COM互操作的技术我们可以Office的自动化操作。

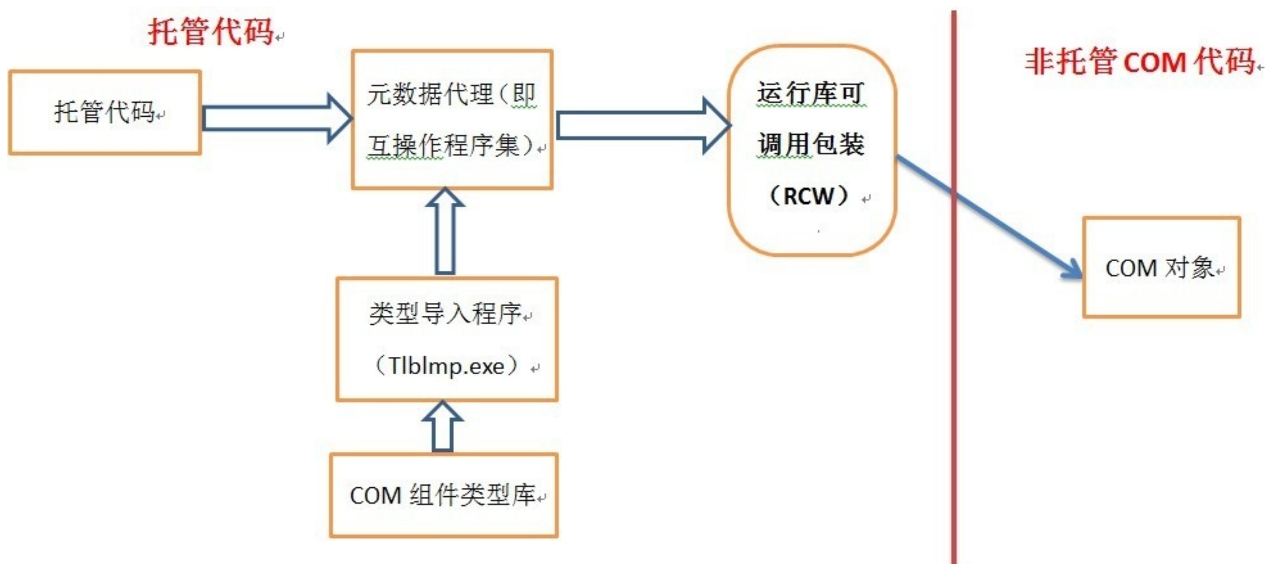
三、在**C#** 中调用**COM**组件的实现原理剖析

通过上面的例子，大家可以看出在.NET 中调用COM组件是非常方便和简单的，以至于我们根本不能明白它背后的原理的，下面就介绍在托管代码中调用COM组件的实现原理和需要的步骤。

要运行上面的程序必须添加一个互操作程序集——“**Microsoft.Office.Interop.Word 14.0.0.0**”，大家可以通过下面的步骤来添加这个程序集：

- References tab." data-guid="e14ff6e02fb2f7ef9fc5a514742ce041"> Solution Explorer, right-click the References folder and then click Add Reference." data-guid="103a54741fa87406ee1f7f3425dd7823">请在“解决方案资源管理器”中，右击“引用”文件夹，然后单击“添加引用”。
- .NET tab, click the most recent version of Microsoft.Office.Interop.Excel." data-guid="eb8f0fe03748066f9280158a455e5259">在**“.NET**”选项卡上，选中最新版本的 **Microsoft.Office.Interop.Word**。 Microsoft.Office.Interop.Excel 14.0.0.0." data-guid="e2d0f21e6a6898f0a89d3bd78687a7fe">例如，“Microsoft.Office.Interop.Excel 14.0.0.0”。 OK." data-guid="d75006f118c9cfc06bd2398b7c6cc909">单击“确定”。

通过上面添加引用的步骤可以看出，**Microsoft.Office.Interop.Word.dll** 是一个.NET程序集，而不是COM组件，这时候朋友们肯定有这样的疑问——不是调用COM组件的吗？怎么在托管代码中调用.NET 程序集的？这样怎么能算是在.NET下调用COM组件的演示了？ 然而事实是—— **Microsoft.Office.Interop.Word.dll** 确实是一个.NET程序集，并且它也叫做COM组件的互操作程序集，这个程序集中包含了COM组件中定义的类型的元数据， 托管代码通过调用互操作程序集中公开的接口或对象来间接地调用COM对象和接口的。由于托管代码中不能直接使用COM对象和接口，所以托管代码对COM对象的调用时是通过CLR的 **COM Interop** 层作为代理完成的，这个代理就是RCW（即Runtime Callable Wrapper,运行时可调用包装），所以对COM对象的调用，都是通过RCW来完成的，RCW做的工作主要有激活COM对象和在托管代码和非托管代码之间进行数据封送处理（从这里可以看出，RCW就是 .NET平台和COM组件之间的一个代理，微软的很多技术都使用了代理的，例如WCF技术——我们在代码中创建的对象其实只是服务的一个代理，通过代理对象来访问真真的对象的服务，即方法。讲到代理的技术，C#中的委托也是代理的一种实现，此时又想到了23中设计模式中的——代理模式，然而生活中也不乏代理的例子，租房中介，代理服务器等）。下面通过一个图来演示下 在**.NET**中调用**COM**组件的原理：



关于通过TlbImp.exe工具来生成互操作程序集步骤，这里我就不多详细诉说了，大家可以参考MSDN中这个工具详细使用说明 ：http://msdn.microsoft.com/zh-cn/library/tt0cf3sx(v=VS.80).aspx.aspx) 。

然而我们也可以使用Visual Studio中内置的支持来完成为COM类型库创建互操作程序集的工作，我们只需要在**VS**中为**.NET** 项目添加对应的**COM**组件的引用，此时**VS**就会自动将**COM**类型库中的**COM**类型库转化为程序集中的元数据，并在项目的Bin目录下生成对于的互操作程序集，所以在VS中添加COM引用，其实最后程序中引用的是互操作程序集，然后通过RCW来对COM组件进行调用。 然而对于Office中的**Microsoft.Office.Interop.Wordd.dll**，这个程序集也是互操作程序集，但是它又是主互操作程序集，即PIA(Primary Interop Assemblies)。主互操作程序集是一个由供应商提供的唯一的程序集，为了生成主互操作程序集，可以在使用TlbImp命令是打开 /primary 选项。看到这里，朋友们肯定有这样的疑问：PIA与普通程序集到底有什么区别呢？——区别就是PIA除了包含了COM组件定义的数据类型外，还包含了一些特殊的信息，如公钥，COM类型库的提供者等信息。然而 为什么需要主互操作程序集的呢 ？ 对于这个问题的答案就是——主互操作程序集可以帮助我们解决部署程序时，引用互操作程序集版本不一致的问题。(如果开发人员会为一个COM组件类型库生成多个互操作程序集，项目中引用的互操作程序集版本与部署时的互操作程序集版本不一致的问题，有了互操作程序集时，我们可以直接引用官方提供主互操作程序集。)

四、错误处理

知道了如何调用COM组件之后，大家或许会问：如果调用COM对象的方法失败时怎么去获取失败的信息呢？对于这个疑问，错误的处理的方法和我们平常托管代码中的处理方式是一样的，下面就具体看看是如何获取错误信息的，下面这段代码的功能是——打开一个现有的Word文档并插入相应的文本，当指定的Word文档不存在时，此时就会出现调用COM对象的Open方法失败的情况，具体代码如下：

```
using System;
using Microsoft.Office.Interop.Word;
using System.IO;
using System.Runtime.InteropServices;
```

```
namespace COM互操作中的错误处理
{
    class Program
    {
        static void Main(string[] args)
        {
           // 打开存在的文档插入文本
            string wordPath = @"D:\test.docx";
            OpenWordDocument(wordPath);
            Console.Read();
        }

        // 向现有文档插入文本
        private static void OpenWordDocument(string wordPath)
        {
            // 启动Word 应用程序
            Application wordApp = new Application() { Visible = tru
            Document wordDoc=null;
            try
            {
                // 如果文档不存在时，就会出现调用COM对象失败的情况
                // 打开Word文档
                wordDoc = wordApp.Documents.Open(wordPath);
                // 向Word中插入文本
                Range wordRange = wordDoc.Range(0, 0);
                wordRange.Text = "这是插入的文本";

                // 保存文档
                wordDoc.Save();
            }
            catch(Exception ex)
            {
                // 获得异常相对应的HRESULT值
                // 因为COM中根据方法返回的HRESULT来判断调用是否成功的
                int HResult = Marshal.GetHRForException(ex);
                // 设置控制台的前景色，即输出文本的颜色
                Console.ForegroundColor = ConsoleColor.Red;
                // 下面把HRESULT值以16进制输出
                Console.WriteLine("调用抛出异常，异常类型为：{0}, HRESU
                Console.WriteLine("异常信息为：" + ex.Message.Replace
            }
            finally
            {
                // 关闭文档并
                if (wordDoc != null)
                {
                    wordDoc.Close();
                }
                // 退出Word程序
                wordApp.Quit();
            }
        }
    }
```

```
}
```

如果我们D盘中不存在一个 test.docx文档时，此时代码就会进入catch块，输出异常信息，运行结果为：



从上面的结果我们看到了一个 HRESULT值，这个值真是COM代码中返回返回的。在COM中，COM方法通过返回 **HRESULT** 来报告错误；.NET 方法则通过引发异常来报告错误，为了方便地在托管代码中获得COM代码中出现的错误和异常信息，CLR提供了两者之间的转换，每一个代表错误发生的HRESULT都会被映射到.NET Framework中的一个异常类，对于具体的映射关系可以参考MSDN中 的文章：http://msdn.microsoft.com/zh-cn/library/9ztbc5s1(VS.80).aspx.aspx)，我这里就不具体用表格列出来的。 如果某个HRESULR不能被映射到等效的.NET Framework异常类时，那么就会被映射到**COMException**异常类， 我们可以通过 **Marshal**类的**GetHRForException**方法来获得异常类对应的HRESULT值（该方法的使用在上面代码中已经贴出）

五、小结

关于在.NET中调用COM组件的介绍就到这里的，即使我们在.NET中调用COM对象的方法是非常的简单和方便，但是理解CLR 为我们背后完成的工作到底有哪些和理解托管代码中调用COM组件原理也是相当有必要的。因为理解了调用的原理之后，当我们出现问题的时候就可以很快找到解决方案并解决它，不会觉得无从下手，这样就可以帮助我们提供解决问题的能力。

CLR

相信大家在面试的时候会经常问到事件和委托的区别，为什么.net中需要事件和委托这样类似的问题吧，对于一些初学者来说可平时用的过程中也不知道为什么，只知道这样用，而对于其中的实现机制不是很清楚，所以面试的时候总是感觉回答的不是很有底气的，对于委托和事件园子里面也有很多人写过这样的文章，比如张子阳博客中C# 中的委托和事件，这篇文章由浅入深讲解了.net中的事件和委托。所以比较建议初学者看看的，而且很容易懂.(本人第一次写，如果什么地方说错了的地方请大家海涵和及时纠正我)

在张子阳的文章我相信已经把事件和委托讲的很清楚了，下面我说说我感觉需要注意的地方。

为什么会有委托

在C++中用函数指针来实现回调函数(回调函数是一种非常有用的编程机制)，然而函数指针不是类型安全的，所以.net Framework提供了称为委托的类型安全的机制来实现函数的回调。

编译器如何解析委托

当我们像下面一样在代码中定义一个委托时，

Public delegate void Comparator(int value);

但是编译器遇到这行代码会定义一个类：

View Code

```
 1 Public class Comparator:System.MulticastDelegate
 2 {
 3         public Comparator(Object object, IntPtr method);
 4
 5         public virtual Void Invoke(Int32 value);
 6
 7         public virtual IAsyncResult BeginInvoke(Int32 vlaue, Asy
 8
 9          public virtual void EndInvoke(IAsyncResult result);
10
11 }
```

从上面代码可以知道 委托 也是一个类，其中有一个构造器，Invoke方法，BeginInvoke方法 和EndInvoke方法。构造器有两个参数，对象引用传给构造器的Object参数，方法的引用传给method 参数，对于静态方法，会为Object 参数传递

null.

事件

先看一个事件的定义：

编译器在编译事件的时候会把它转换为三个构造：

```
        private Comparator onComparator = null;

        public void add_onComparator(Comparator value)
        {
            // 以一种线程安全的方式对事件添加一个委托
      }
        public void remove_onComparator(Comparator value)
        {
            // 以一种线程安全的方式对事件移出一个委托
      }
```

从上面代码可以看出第一部分是申明一个私有的委托字段， 后面两部分是对这个委托字段的add访问器和remove 访问器，我们知道属性中有get 和set 访问器，其实事件就是委托字段的访问器，只是访问器方法用add 和remove,而属性用get和set.

总结：

到这里我要讲的差不多说完了， 这是我第一次写文章，尽管上面的内容理解的不是很深入，但是我只是想通过这样的方式来巩固自己看到的知识，因为我觉得这样可以记录下我不同时段对知识的理解以及写的时候自己也在不断思考，这样会有利于对知识的理解。

最后我为初学者推荐关于深入理解.net Framework几本书， 因为我感觉很多初学者不知道买什么书来学习。

1. CLR via C# (第三版) 作者：Jeffrey Richter (周靖译) 清华大学出版社
2. 深入理解C#（第2版）作者：Jon Skeet 周靖（译）人民邮电出版社

现在关于C# 方面的书籍很多，所以对于一些初学者来说不知道怎么选择， 我推荐上面两本书，如果认真的看完的话， 我相信你肯定对.net 会有一定的理解， 然后通过项目实践的方式对书中内容进行巩固。个人觉得要深入理解程序底层的东西，有必要阅读一些关于操作系统和编译器相关的书籍，本人一向提倡“知其然知其所以然”的学习方式。

在此推荐一本操作系统相关的书籍： 深入理解计算机系统 （美）布莱恩特，奥哈拉伦 著 龚奕利，雷迎春 译 机械工业出版社。

希望这篇文章对大家会有帮助。

作者：Learning hard

出处：http://www.cnblogs.com/zhili/

谈谈: String 和StringBuilder区别和选择

对于string 和stringbuilder相信大家经常会使用到， 但是相信它们的区别和如何选择对于初学者还是会有不清楚的，下面我来分享下我的理解， 如果什么不对的地方希望大家指出来。

（一）String 和StringBuilder区别

1. 构造字符串

在C#中， 不能使用new 操作符从一个文本常量字符串构造一个String 对象，因为String 类中没有提供接受字符串的参数的构造函数。

```
string str = " Hello World"; // 对的
string str2 = new string("Hello");//错误。
```

这时候通过ldstr(Load string)指令来创建一个String 对象的， 而不是用newobj创建对象实例的。

2. String 对象是不可变的， 具体指字符串一旦创建了， 就不能更改、不能变长或变短。 主要是因为 String 中的索引器是只读的， 因为String是不可变的，这就使得在操作或访问一个字符串时不会发生线程同步问题。

String类中索引器定义：

有些朋友对于String 不可变 有一些误解， 可能因为下面的例子：

有些朋友可能认为String对象str被修改了， 其实并不是这样的，String对象str已经重新指向了一个新的字符串常量："Hello", 而不是在原来字符串上修改， 这时候因为"Hello World"因为没有引用了，所以会认为是垃圾，会被垃圾回收。

String 字符串中还有一个 字符串留用（string interning）技术， 在这里我就不介绍了，想了解的朋友可以查看Artech 博客中的字符串驻留这篇文章。

3. 而StringBuilder是可变的， 可利用它高效地对字符串和字符进行动态处理。可以通过Append和Insert方法等方法来更改字符数组的内容，而不会造成在托管堆上分配新对象。

（二）什么时候用String, 而什么时候用该用StringBuilder

有些人可能会认为既然这样， 那是不是不需要String类型的？ 只要我们在所有需要用String的地方都用StringBuilder代替就可以了， 答案肯定是否定的，

我个人理解是：当要对字符串进行频繁的操作的时候，在 String 和 StringBuilder 之间，我们应该选择 StringBuilder， 对于一般的操作操作还是使用String类型，因为stringbuilder功能强大，这意味着其底层实现更复杂，一些简单的功能用string当然更简洁、甚至比用Stringbuilder更高效率。

如果需要转载的朋友请注明出处。

谈谈：程序集加载和反射

最近一直都在看关于程序集加载和反射方面的资料， 所以在这里把我所学习到的东西记录下来，方便自己以后复习，也给园子里面不懂的朋友参考。

一、程序集的加载

JIT编译器器器将IL代码编译成本地代码时， 会查看IL代码中引用了哪些类型。在运行过程中，JIT编译器利用程序集的TypeRef和AssemblyRef元数据表来确定哪一个程序集定义了所引用的类型，然后JIT编译器将对应程序集加载到AppDomain中，在内部,CLR使用System.Reflection.Assembly类的静态方法Load来尝试加载一个程序集。然而如果我们想动态加载一个程序集时，可以使用Assembly的Load方法来动态加载程序集，其中Assembly类中还提供了其他的加载程序集方法，有LoadFrom(string path), LoadFile(stringassemblyFile)等，具体方法的使用和解释可以参照MSDN中的介绍：http://msdn.microsoft.com/zh-cn/library/xbe1wdx9

二、反射机制

.net中反射在运行中过程中解析程序集中的元数据，获得类型中的成员（包括字段、构造器、方法、属性、事件等）信息。

动态加载一个程序集并获得类型中的成员

把下面的类放在一个类库工程中，并编译生成程序集（例如为ClassLibrary1.dll,假设把dll放在D盘根目录下面）

View Code

```
1   public class ReflectTestClass
2      {
3         public  string name;
4         public int age;
5         public string Name
6         {
7             get { return name; }
8             set { name = value; }
9         }
10
11        public int Age
12        {
13            get { return age; }
14            set { age = value; }
15        }
16
17         /// <summary>
18         /// No Paramter Constructor
19         /// </summary>
20        public ReflectTestClass()
21        {
22        }
23
24         /// <summary>
25         /// Constructor with Parameter
26         /// </summary>
27         /// <param name="name"></param>
28         /// <param name="age"></param>
29         public ReflectTestClass(string names,int ages)
30         {
31             this.name = names;
32             this.age = ages;
33         }
34
35         public string writeString(string name)
36         {
37             return "Welcome " + name;
38         }
39
40         public static string WriteName(string name)
41         {
42             return "Welcome "+name +" Come here";
43         }
44
45         public string WirteNopara()
46         {
47             return "The method is no parameter ";
48         }
49     }
```

然后建立一个控制台程序用来动态加载上面生成的程序集和输出类型中的成员，代码中有详细的介绍。

```
class Program
    {
        static void Main(string[] args)
        {
            Assembly ass;
            Type[] types;
            Type typeA;
            object obj;
            try
            {
                // 从本地中 加载程序集 然后从程序集中通过反射获得类型的信息的
                ass = Assembly.LoadFrom(@"D:\ClassLibrary1.dll");
                types = ass.GetTypes();
                foreach (Type type in types)
                {
                    Console.WriteLine("Class Name is " + type.FullN
                    Console.WriteLine("Constructor Information");
                    Console.WriteLine("-----------------------");
                    // 获取类型的结构信息
                    ConstructorInfo[] myconstructors = type.GetCons
                    ShowMessage<ConstructorInfo>(myconstructors);

                    Console.WriteLine("Fields Information");
                    Console.WriteLine("-----------------------");
                    // 获取类型的字段信息
                    FieldInfo[] myfields = type.GetFields();
                    ShowMessage<FieldInfo>(myfields);

                    Console.WriteLine("All Methods Information");
                    Console.WriteLine("-----------------------");
                    // 获取方法信息
                    MethodInfo[] myMethodInfo = type.GetMethods();
                    ShowMessage<MethodInfo>(myMethodInfo);

                    Console.WriteLine("All Properties Information")
                    Console.WriteLine("-----------------------");
                    // 获取属性信息
                    PropertyInfo[] myproperties = type.GetPropertie
                    ShowMessage<PropertyInfo>(myproperties);
                }

                // 用命名空间+类名获取类型
                typeA = ass.GetType("ClassLibrary1.ReflectTestClass

                // 获得方法名称

                MethodInfo method = typeA.GetMethod("writeString");

                // 创建实例
```

```
                obj = ass.CreateInstance("ClassLibrary1.ReflectTest

                string result = (String)method.Invoke(obj,new strin
                Console.WriteLine("Invoke Method With Parameter");
                Console.WriteLine("-----------------------");
                Console.WriteLine(result);
                Console.WriteLine("-----------------------");
                Console.WriteLine();
                method = typeA.GetMethod("WriteName");
                result = (string)method.Invoke(null,new string[] {'
                Console.WriteLine("Invoke Static Method with Parame
                Console.WriteLine("-----------------------");
                Console.WriteLine(result);
                Console.WriteLine("-----------------------");
                Console.WriteLine();
                method = typeA.GetMethod("WirteNopara");
                Console.WriteLine("Invoke Method with NOParameter")
                result = (string)method.Invoke(obj, null);
                Console.WriteLine("-----------------------");
                Console.WriteLine(result);
                Console.WriteLine("-----------------------");
            }

            catch(FileNotFoundException ex)
            {
                Console.WriteLine(ex.Message);
            }

            Console.ReadLine();
        }

        /// <summary>
        /// 显示数组信息
        /// </summary>
        /// <typeparam name="T"></typeparam>
        /// <param name="os"></param>
        public static void ShowMessage<T>(T[] array)
        {
            foreach(T member in array)
            {
                Console.WriteLine(member.ToString());
            }

            Console.WriteLine("-----------------------");
            Console.WriteLine();
        }
    }
```

筛选返回的成员种类

可以调用Type的GetMembers,GetFields,GetMethods,GetProperties或者GetEvenents方法来查询一个类型的成员。在调用上面的任何一个方法时，都可以传递System.Reflection.BindingFlags枚举类型的一个实例，使用这个枚举类型目的是对这些方法返回的成员进行筛选。对于这个枚举类型中成员的信息可以参考MSDN:http://msdn.microsoft.com/zh-cn/library/system.reflection.bindingflags(v=VS.80).aspx.aspx)

注意：在返回一个成员集合的所有方法中，都有一个不获取任何实参的重载版本。如果不传递BindingFlags实参，所有这些方法都返回公共成员，默认设置为**BindingFlags.Public|BindingFlags.Instance|BindingFlags.Static**. (如果指定Public或NonPublic,那么必须同时指定Instance,否则不返回成员)。

利用反射获得委托和事件以及创建委托实例和添加事件处理程序

最近一些都在看关于反射的内容，然后在网上大多数都是通过反射获得类型中方法，属性、字段这样的文章， 但是对于如何获得委托类型怎么去实现的却没有，所以写下这边篇文章来让自己以后很好的复习以及想了解的朋友做参考。

一、 利用反射获得委托类型并创建委托实例

```
using System;
using System.Reflection;

namespace ConsoleApplication1
{

    public  class Test
    {

        public delegate void delegateTest(string s);
        public void method1(string s)
        {
            Console.WriteLine("Create Delegate Instance: " + s);
        }

    }

    class Program
    {
        static void Main(string[] args)
        {
            Test test = new Test();
            Type t = Type.GetType("ConsoleApplication1.Test");
            // 因为委托类型编译后是作为类的嵌套类型的，所以这里通过GetNeste
            Type nestType = t.GetNestedType("delegateTest");

            MethodInfo method =test.GetType().GetMethod("method1",
            if (method != null)
            {
                // 创建委托实例
                Delegate method1 = Delegate.CreateDelegate(nestType
                //动态调用委托实例
                method1.DynamicInvoke("Hello");
            }

            Console.Read();
        }
    }
}
```

二、 利用反射获得事件类型和绑定事件处理程序

```
using System;
using System.Reflection;

namespace ConsoleApplication2
{
    public class Test
    {
        public event EventHandler TestEvent;
        public void Triggle()
        {
            if (TestEvent != null)
            {
                TestEvent(this, null);
            }
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Test testT=new Test();
            EventInfo eventinfo = typeof(Test).GetEvent("TestEvent'
            if (eventinfo != null)
            {
                // 为事件动态绑定处理程序
                eventinfo.AddEventHandler(testT, new EventHandler(t
                testT.Triggle();
            }
            Console.Read();

        }
        public static void triggleEvent(object sender, EventArgs e)
        {
            Console.WriteLine("Event has been Triggled");
        }
    }
}
```

希望这些使大家对放射有个更好的理解。

谈谈：.Net中的序列化和反序列化

序列化和反序列化相信大家都经常听到，也都会用， 然而有些人可能不知道：.net为什么要有这个东西以及.net Frameword如何为我们实现这样的机制， 在这里我也是简单谈谈我对序列化和反序列化的一些理解。

一、什么序列化和反序列化

序列化通俗地讲就是将一个对象转换成一个字节流的过程，这样就可以轻松保存在磁盘文件或数据库中。反序列化是序列化的逆过程，就是将一个字节流转换回原来的对象的过程。

然而为什么需要序列化和反序列化这样的机制呢？这个问题也就涉及到序列化和反序列化的用途了，

对于序列化的主要用途有：

- 将应用程序的状态保存在一个磁盘文件或数据库中，并在应用程序下次运行时恢复状态。例如, Asp.net 中利用序列化和反序列化来保存和恢复会话状态。
- 一组对象可以轻松复制到Windows 窗体的剪贴板中，再粘贴回同一个或者另一个应用程序。
- 将对象按值从一个应用程序域中发送到另一个程序域

并且如果把对象序列化成内存中的字节流，就可以利用一些其他的技术来处理数据，例如，对数据进行加密和压缩等。

二、序列化和反序列简单使用

.Net Framework 提供二种序列化方式：

- 二进制序列化
- XML 和SOAP序列化

序列化和反序列化的简单使用：

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

namespace Serializable
{
    [Serializable]
    public class Person
    {
        public string personName;
```

```
        [NonSerialized]
        public string personHeight;

        private int personAge;
        public int PersonAge
        {
            get { return personAge; }
            set { personAge = value; }
        }

        public void Write()
        {
            Console.WriteLine("Person Name: "+personName);
            Console.WriteLine("Person Height: " +personHeight);
            Console.WriteLine("Person Age: "+ personAge);
        }

    }
    class Program
    {
        static void Main(string[] args)
        {
            Person person = new Person();
            person.personName = "Jerry";
            person.personHeight = "175CM";
            person.PersonAge = 22;
            Stream stream = Serialize(person);

            //为了演示，都重置
            stream.Position = 0;
            person = null;

            person = Deserialize(stream);
            person.Write();
            Console.Read();

        }
        private static MemoryStream Serialize(Person person)
        {
            MemoryStream stream = new MemoryStream();

            // 构造二进制序列化格式器
            BinaryFormatter binaryFormatter = new BinaryFormatter()
            // 告诉序列化器将对象序列化到一个流中
            binaryFormatter.Serialize(stream, person);

            return stream;

        }

        private static Person Deserialize(Stream stream)
        {
            BinaryFormatter binaryFormatter = new BinaryFormatter()
```

```
                return (Person)binaryFormatter.Deserialize(stream);
            }

        }
    }
```

主要是调用System.Runtime.Serialization.Formatters.Binary命名空间下的BinnaryFormatter类来进行序列化和反序列化，调用反序列化后的结果截图：

file:///c:/users/v-tozhi/documents/visual studio 2010/Projects/Serializable/Serializable/bin/Debug/...

```
Person Name: Jerry
Person Height:
Person Age: 22
```

从中可以看出除了标记NonSerialized的其他成员都能序列化,注意这个属性只能应用于一个类型中的字段，而且会被派生类型继承。

SOAP 和XML 的序列化和反序列化和上面类似，只需要改下格式化器就可以了，这里我就不列出来了。

三、控制序列化和反序列化

有两种方式来实现控制序列化和反序列化：

- 通过**OnSerializing, OnSerialized,OnDeserializing, OnDeserialized,NonSerialized**和**OptionalField**等属性
- 实现**System.Runtime.Serialization.ISerializable**接口

第一种方式实现控制序列化和反序列化代码：

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

namespace ControlSerialization
{
    [Serializable]
    public class Circle
    {
        private double radius; //半径
        [NonSerialized]
        public double area; //面积
```

```
        public Circle(double inputradiu)
        {
            radius = inputradiu;
            area = Math.PI * radius * radius;
        }

        [OnDeserialized]
        private void OnDeserialized(StreamingContext context)
        {
            area = Math.PI * radius * radius;
        }

        public void Write()
        {
            Console.WriteLine("Radius is: " + radius);
            Console.WriteLine("Area is: " + area);
        }
    }
    class Program
    {

        static void Main(string[] args)
        {
            Circle c = new Circle(10);
            MemoryStream stream =new MemoryStream();
            BinaryFormatter formatter = new BinaryFormatter();
            // 将对象序列化到内存流中，这里可以使用System.IO.Stream抽象类
            formatter.Serialize(stream,c);
            stream.Position = 0;
            c = null;
            c = (Circle)formatter.Deserialize(stream);
            c.Write();
            Console.Read();

        }
    }
}
```

运行结果为：

```
file:///c:/users/v-tozhi/documents/visual studio 2010/Projects/Serializable/ControlSerialization/bin...
Radius is: 10
Area is: 314.159265358979
```

注意：如果注释掉 **OnDeserialized**属性的话，area字段的值就是0了，因为area字段没有被序列化到流中。

在上面需要序列化的对象中，格式化器只会序列化对象的radius字段的值。area字段中的值不会序列化，因为该字段已经应用了NonSerializedAttribute属性，然后我们用Circle c=new Circle(10)这样代码构建一个Circle对象时，在内部，area会设置一个约为314.159这样的值，这个对象序列化时，只有radius的字段的值（10）写入流中， 但当反序列化成一个Circle对象时，它的area字段的值会初始化为0，而不是约314.159的一个值。为了解决这样的问题，所以自定义一个方法应用**OnDeserializedAttribute**属性。此时的执行过程为：每次反序列化类型的一个实例，格式化器都会检查类型中是否定义了 一个应用了该attribute的方法，如果是，就调用该方法，调用该方法时，所有可序列化的字段都会被正确设置。除了**OnDeserializedAttribute**这个定制attribute,system.Runtime.Serialization命名空间还定义了**OnSerializingAttribute,OnSerializedAttribute**和**OnDeserializingAttribute**这些定制属性。

实现**ISerializable**接口方式控制序列化和反序列化代码：

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;
using System.Security.Permissions;

namespace ControlSerilization2
{
    [Serializable]
    public class MyObject : ISerializable
    {
        public int n1;
        public intn2;

        [NonSerialized]
        public String str;

        public MyObject()
        {
        }

        protected MyObject(SerializationInfo info, StreamingContext
        {
            n1 = info.GetInt32("i");
            n2 = info.GetInt32("j");
            str = info.GetString("k");
        }

        [SecurityPermissionAttribute(SecurityAction.Demand, Serial
        public virtual void GetObjectData(SerializationInfo info, S
        {
            info.AddValue("i", n1);
            info.AddValue("j", n2);
```

```
            info.AddValue("k", str);
        }

        public void Write()
        {
            Console.WriteLine("n1 is: " + n1);
            Console.WriteLine("n2 is: " + n2);
            Console.WriteLine("str is: " + str);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            MyObject obj = new MyObject();
            obj.n1 = 2;
            obj.n2 = 3;
            obj.str = "Jeffy";
            MemoryStream stream = new MemoryStream();
            BinaryFormatter formatter = new BinaryFormatter();
            // 将对象序列化到内存流中，这里可以使用System.IO.Stream抽象类
            formatter.Serialize(stream, obj);
            stream.Position = 0;
            obj = null;
            obj = (MyObject)formatter.Deserialize(stream);
            obj.Write();
            Console.Read();
        }
    }
}
```

结果为：

```
file:///c:/users/v-tozhi/documents/visual studio 2010/Projects/Serializable/Contr
n1 is: 2
n2 is: 3
str is: Jeffy
```

此时的执行过程为：当格式化器序列化对象时，会检查每个对象，如果发现一个对象的类型实现了**ISerializable**接口，格式化器会忽视所有定制属性，改为构造一个新的**System.Runtime.Serialization.SerializationInfo**对象，这个对象包含了要实际为对象序列化的值的集合。构造好并初始化好SerializationInfo对象后，格式化器调用类型的**GetObjectData**方法，并向它传递对**SerializationInfo**对象的引用，**GetObjectData**方法负责决定需要哪些信息来序列化对象，并将这些信息添加到**SerializationInfo**对象中，通过调用AddValue方法来添加需要的每个数据，添加好所有必要的序列化信息后，会返回至格式化器，然后格式化器获取已经添加

到**SerializationInfo**对象中的所有值，并将它们都序列化到流中，当反序列化时，格式化器从流中提取一个对象时，会为新对象分配内存，最初，这个对象的所有字段都设为0或null,然后，格式化器检查类型是否实现了**ISerializable**接口，如果存在这个接口， 格式化器就尝试调用一个特殊构造器，它的参数和**GetObjectData**方法的完全一致。

四、格式化器如何序列化和反序列化

从上面的分析中可以看出，进行序列化和反序列化主要是格式化器在工作的，然而下面就是要讲讲格式化器是如何序列化一个应用了 **SerializableAttribute** 属性的对象。

1. 格式化器调用**FormatterServices**的**GetSerializableMembers**方法：public static MemberInfo[] GetSerializableMembers(Type type,StreamingContext context);这个方法利用发射获取类型的public和private实现字段（标记了**NonSerializedAttributee**属性的字段除外）。方法返回由MemberInfo对象构成的一个数组，其中每个元素对应于一个可序列化的实例字段。
2. 对象被序列化，**System.Reflection.MemberInfo**对象数组传给**FormatterServices**的静态方法**GetObjectData**: public static object[] GetObjectData(Object obj,MemberInfo[] members); 这个方法返回一个**Object**数组，其中每个元素都标识了被序列化的那个对象中的一个字段的值。
3. 格式化器将程序集标识和类型的完整名称写入流中。
4. 格式化器然后遍历两个数组中的元素，将每个成员的名称和值写入流中。

接下来是解释格式化器如何自动反序列化一个应用了 **SerializableAttribute**属性的对象。

1. 格式化器从流中读取程序集标识和完整类型名称。
2. 格式化器调用**FormatterServices**的静态方法**GetUninitializedObject**: public static Object GetUninitializedObject(Type ttype);这个方法为一个新对象分配内存，但不为对象调用构造器。然而，对象的所有字段都被初始化为0或null.
3. 格式化器现在构造并初始化一个**MemberInfo**数组，调用**FormatterServices**的**GetSerializableMembers**方法，这个方法返回序列化好、现在需要反序列化的一组字段。
4. 格式化器根据流中包含的数据创建并初始化一个**Object**数组。
5. 将对新分配的对象、**MemberInfo**数组以及并行**Object**数组的引用传给**FormatterServices**的静态方法**PopulateObjectMembers**:

public static Object PopulateObjectMembers(Object obj,MemberInfo[] members,Object[] data);这个方法遍历数组，将每个字段初始化成对应的值。

注：格式化如何序列化和反序列对象部分摘自CLR via C#(第三版)，写在这里可以让初学者进一步理解格式化器在序列化和反序列化过程中所做的工作。

写到这里这篇关于序列化和反序列的文章终于结束了， 希望对自己以后复习和园子里的朋友有帮助。

C#设计模式

[转发]UML类图符号 各种关系说明以及举例

UML中描述对象和类之间相互关系的方式包括：依赖（Dependency），关联（Association），聚合（Aggregation），组合（Composition），泛化（Generalization），实现（Realization）等。

- 依赖**（**Dependency**）**：元素A的变化会影响元素B，但反之不成立，那么B和A的关系是依赖关系，B依赖A；类属关系和实现关系在语义上讲也是依赖关系，但由于其有更特殊的用途，所以被单独描述。uml中用带箭头的虚线表示Dependency关系，箭头指向被依赖元素。
- 泛化（**Generalization**）：通常所说的继承（特殊个体 is kind of 一般个体）关系，不必多解释了。uml中用带空心箭头的实线线表示Generalization关系，箭头指向一般个体。
- 实现（**Realize**）：元素A定义一个约定，元素B实现这个约定，则B和A的关系是Realize，B realize A。这个关系最常用于接口。uml中用空心箭头和虚线表示Realize关系，箭头指向定义约定的元素。
- 关联（**Association**）：元素间的结构化关系，是一种弱关系，被关联的元素间通常可以被独立的考虑。uml中用实线表示Association关系，箭头指向被依赖元素。
- 聚合（**Aggregation**）：关联关系的一种特例，表示部分和整体（整体 has a 部分）的关系。uml中用带空心菱形头的实线表示Aggregation关系，菱形头指向整体。
- 组**合（**Composition**）**：组合是聚合关系的变种，表示元素间更强的组合关系。如果是组合关系，如果整体被破坏则个体一定会被破坏，而聚合的个体则可能是被多个整体所共享的，不一定会随着某个整体的破坏而被破坏。uml中用带实心菱形头的实线表示Composition关系，菱形头指向整体。

其中依赖（Dependency）的关系最弱，而关联（Association），聚合（Aggregation），组合（Composition）表示的关系依次增强。换言之关联，聚合，组合都是依赖关系的一种，聚合是表明对象之间的整体与部分关系的关联，而组合是表明整体与部分之间有相同生命周期关系的聚合。

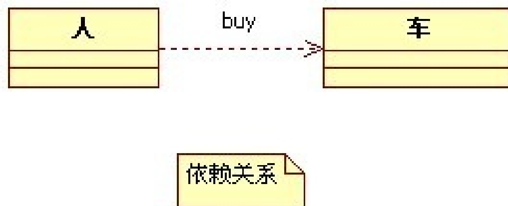
而关联与依赖的关系用一句话概括下来就是，依赖描述了对象之间的调用关系，而关联描述了对象之间的结构关系。

后面的例子将针对某个具体目的来独立地展示各种关系。虽然语法无误，但这些例子可进一步精炼，在它们的有效范围内包括更多的语义。

1.1.1 依赖（Dependency）:虚线箭头表示

1、依赖关系也是类与类之间的联结 2、依赖总是单向的。（#add 注意，要避免双向依赖。一般来说，不应该存在双向依赖。） 3、依赖关系在 Java 或 C++ 语言中体现为局部变量、方法的参数或者对静态方法的调用。

```
class Person
{
    void buy(Car car)
    {
        ...
    }
}
```



表示方法：虚线加箭头

特点：当类与类之间有使用关系时就属于依赖关系，不同于关联关系，依赖不具有"拥有关系"，而是一种"相识关系"，只在某个特定地方（比如某个方法体内）才有关系

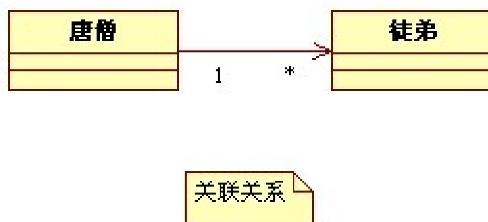
1.1.2 关联（Association）：实线箭头表示

1、关联关系是类与类之间的联结，它使一个类知道另一个类的属性和方法。 2、关联可以是双向的，也可以是单向的（#add还有自身关联）。双向的关联可以有两个箭头或者没有箭头，单向的关联有一个箭头。 3、在 Java 或 c++ 中，关联关系是通过使用成员变量来实现的。

```
class 徒弟
{

};

class 唐僧
{
    protected: list<徒弟> tdlist;
};
```



表示方法：实线箭头

特征：表示类与类或类与接口之间的依赖关系，表现为“拥有关系”；具体到代码可以用实例变量来表示。

1.1.3 聚合（Aggregation）：带空心菱形头表示

1、聚合关系是关联关系的一种，是强的关联关系。 2、聚合是整体和部分之间的关系，例如汽车由引擎、轮胎以及其它零件组成。 3、聚合关系也是通过成员变量来实现的。但是，关联关系所涉及的两个类处在同一个层次上，而聚合关系中，两个类处于不同的层次上，一个代表整体，一个代表部分。 4、关联与聚合仅仅从 Java 或 C++ 语法上是无法分辨的，必须考察所涉及的类之间的逻辑关系。

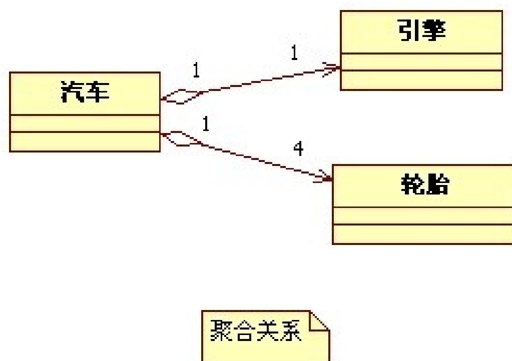
```
class 引擎
{

};

class 轮胎
{

};

class 汽车
{
    protected:引擎 engine;
    protected:轮胎 tyre[4];
};
```



表示方法：空心菱形头

特征：属于是关联的特殊情况，体现部分-整体关系，是一种弱拥有关系；整体和部分可以有不一样的生命周期；是一种弱关联；

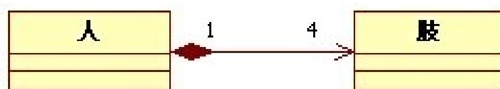
1.1.4 合成（Composition）：带实心菱形头的实线表示

1、合成关系是关联关系的一种，是比聚合关系还要强的关系。 2、它要求普通的聚合关系中代表整体的对象负责代表部分的对象的生命周期。

```
class 肢
{

};

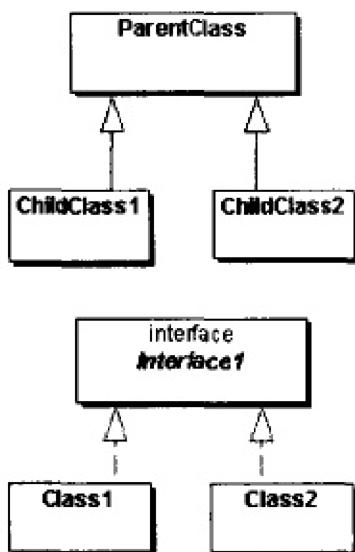
class 人
{
    protected: 肢 limb[4];
};
```



一般是实心菱形加实线箭头表示

特征：属于是关联的特殊情况，也体现了体现部分-整体关系，是一种强“拥有关系”；整体与部分有相同的生命周期，是一种强关联；

一般化关系(泛化和实现)：表示类与类之间的继承关系，接口与接口之间的继承关系，或类对接口的实现关系。一般化关系是子类指向父类的，或从实现接口的类指向被实现的接口，与继承或实现的方向相反。如下图所示：



图：一般化关系

1.1.5 泛化（Generalization）：

带空心箭头的实线线表示

泛化（图**H**）表示一个更泛化的元素和一个更具体的元素之间的关系。泛化是用于对继承进行建模的UML元素。在Java中，用*extends*关键字来直接表示这种关系。

泛化关系表示类与类之间的继承关系，接口与接口之间的继承关系。图H

Java	UML

public abstract class Employee {
 	
}
public class Professor extends Employee {
 	
}
	Employee ◁— Professor

1.1.6 实现（Realization）：空心箭头和虚线表示

实例（图**I**）关系指定两个实体之间的一个合同。换言之，一个实体定义一个合同，而另一个实体保证履行该合同。对Java应用程序进行建模时，实现关系可直接用*implements*关键字来表示。

图I

Java	UML

public interface CollegePerson {
 	
}
public class Professor implements CollegePers
 	
}
	CollegePerson ◁- - - Professor

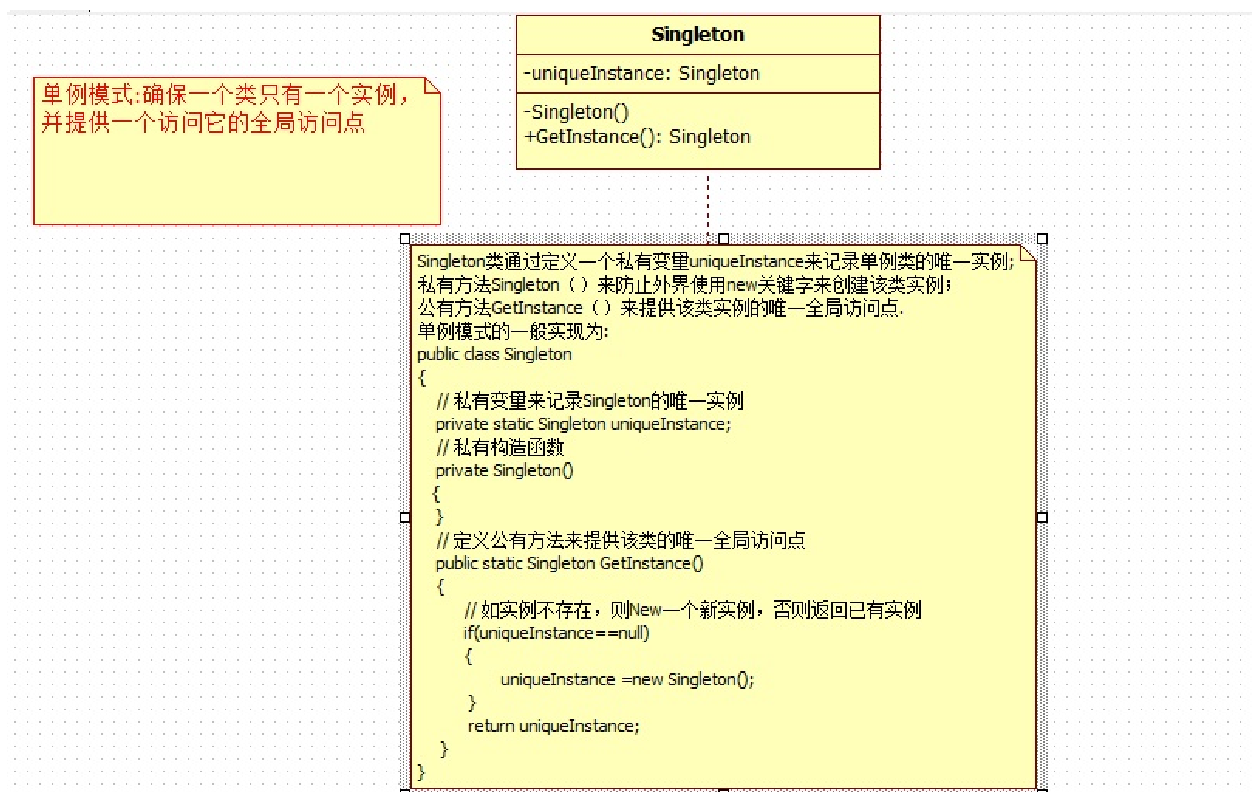
C#设计模式(1)——单例模式

一、引言

最近在设计模式的一些内容，主要的参考书籍是《Head First 设计模式》，同时在学习过程中也查看了很多博客园中关于设计模式的一些文章的，在这里记录下我的一些学习笔记，一是为了帮助我更深入地理解设计模式，二同时可以给一些初学设计模式的朋友一些参考。首先我介绍的是设计模式中比较简单的一个模式——单例模式（因为这里只牵涉到一个类）

二、单例模式的介绍

说到单例模式,大家第一反应应该就是——什么是单例模式？，从“单例”字面意思上理解为——一个类只有一个实例，所以单例模式也就是保证一个类只有一个实例的一种实现方法罢了(设计模式其实就是帮助我们解决实际开发过程中的方法, 该方法是为了降低对象之间的耦合度,然而解决方法有很多种,所以前人就总结了一些常用的解决方法为书籍,从而把这本书就称为设计模式)，下面给出单例模式的一个官方定义：确保一个类只有一个实例,并提供一个全局访问点。为了帮助大家更好地理解单例模式,大家可以结合下面的类图来进行理解,以及后面也会剖析单例模式的实现思路:



三、为什么会有单例模式

看完单例模式的介绍,自然大家都会有这样一个疑问——为什么要有单例模式的？它在什么情况下使用的？从单例模式的定义中我们可以看出——单例模式的使用自然是当我们的系统中某个对象只需要一个实例的情况，例如:操作系统中只能有一个任务管理器,操作文件时,同一时间内只允许一个实例对其操作等,既然现实生活中有这样的应用场景,自然在软件设计领域必须有这样的解决方案了(因为软件设计也是现实生活中的抽象)，所以也就有了单例模式了。

四、剖析单例模式的实现思路

了解完了一些关于单例模式的基本概念之后，下面就为大家剖析单例模式的实现思路的，因为在我自己学习单例模式的时候，咋一看单例模式的实现代码确实很简单，也很容易看懂，但是我还是觉得它很陌生（这个可能是看的少的，或者自己在写代码中也用的少的缘故），而且心里总会这样一个疑问——为什么前人会这样去实现单例模式的呢？他们是如何思考的呢？后面经过自己的琢磨也就慢慢理清楚单例模式的实现思路了，并且此时也不再觉得单例模式陌生了，下面就分享我的一个剖析过程的：

我们从单例模式的概念（确保一个类只有一个实例,并提供一个访问它的全局访问点）入手，可以把概念进行拆分为两部分：（1）确保一个类只有一个实例；（2）提供一个访问它的全局访问点；下面通过采用两人对话的方式来帮助大家更快掌握分析思路：

菜鸟：怎样确保一个类只有一个实例了？

老鸟：那就让我帮你分析下，你创建类的实例会想到用什么方式来创建的呢？

新手：用new关键字啊，只要new下就创建了该类的一个实例了，之后就可以使用该类的一些属性和实例方法了

老鸟：那你想过为什么可以使用new关键字来创建类的实例吗？

菜鸟：这个还有条件的吗？........., 哦，我想起来了，如果类定义私有的构造函数就不能在外界通过new创建实例了（注：有些初学者就会问，有时候我并没有在类中定义构造函数为什么也可以使用new来创建对象，那是因为编译器在背后做了手脚了，当编译器看到我们类中没有定义构造函数，此时编译器会帮我们生成一个公有的无参构造函数）

老鸟：不错，回答的很对，这样你的疑惑就得到解答了啊

菜鸟：那我要在哪里创建类的实例了？

老鸟：你傻啊，当然是在类里面创建了（注：这样定义私有构造函数就是上面的一个思考过程的，要创建实例，自然就要有一个变量来保存该实例把，所以就有了私有变量的声明,但是实现中是定义静态私有变量,朋友们有没有想过**——**这里为什么定义为静态的呢**?**对于这个疑问的解释为：每个线程都有自己的线程栈，定义为静态主要是为了在多线程确保类有一个实例）

菜鸟：哦，现在完全明白了，但是我还有另一个疑问——现在类实例创建在类内部，那外界如何获得该的一个实例来使用它了？

老鸟：这个，你可以定义一个公有方法或者属性来把该类的实例公开出去了（注：这样就有了公有方法的定义了，该方法就是提供方法问类的全局访问点）

通过上面的分析，相信大家也就很容易写出单例模式的实现代码了，下面就看看具体的实现代码（看完之后你会惊讶道：真是这样的！）：

```
/// <summary>
  /// 单例模式的实现
  /// </summary>
  public class Singleton
  {
      // 定义一个静态变量来保存类的实例
      private static Singleton uniqueInstance;

      // 定义私有构造函数，使外界不能创建该类实例
      private Singleton()
      {
      }

      /// <summary>
      /// 定义公有方法提供一个全局访问点,同时你也可以定义公有属性来提供全局
      /// </summary>
      /// <returns></returns>
      public static Singleton GetInstance()
      {
          // 如果类的实例不存在则创建，否则直接返回
          if (uniqueInstance == null)
          {
              uniqueInstance = new Singleton();
          }
          return uniqueInstance;
      }
  }
```

上面的单例模式的实现在单线程下确实是完美的,然而在多线程的情况下会得到多个Singleton实例,因为在两个线程同时运行GetInstance方法时，此时两个线程判断(uniqueInstance ==null)这个条件时都返回真，此时两个线程就都会创建Singleton的实例，这样就违背了我们单例模式初衷了，既然上面的实现会运行多个线程执行，那我们对于多线程的解决方案自然就是使**GetInstance**方法在同一时间只运行一个线程运行就好了，也就是我们线程同步的问题了(对于线程同步大家也可以参考我线程同步的文章),具体的解决多线程的代码如下:

```
/// <summary>
    /// 单例模式的实现
    /// </summary>
    public class Singleton
    {
        // 定义一个静态变量来保存类的实例
        private static Singleton uniqueInstance;

        // 定义一个标识确保线程同步
        private static readonly object locker = new object();

        // 定义私有构造函数，使外界不能创建该类实例
        private Singleton()
        {
        }

        /// <summary>
        /// 定义公有方法提供一个全局访问点,同时你也可以定义公有属性来提供全局
        /// </summary>
        /// <returns></returns>
        public static Singleton GetInstance()
        {
            // 当第一个线程运行到这里时，此时会对locker对象 "加锁",
            // 当第二个线程运行该方法时，首先检测到locker对象为"加锁"状态,
            // lock语句运行完之后（即线程运行完之后）会对该对象"解锁"
            lock (locker)
            {
                // 如果类的实例不存在则创建，否则直接返回
                if (uniqueInstance == null)
                {
                    uniqueInstance = new Singleton();
                }
            }

            return uniqueInstance;
        }
    }
```

上面这种解决方案确实可以解决多线程的问题,但是上面代码对于每个线程都会对线程辅助对象**locker**加锁之后再判断实例是否存在，对于这个操作完全没有必要的，因为当第一个线程创建了该类的实例之后，后面的线程此时只需要直接判断（**uniqueInstance==null**）为假，此时完全没必要对线程辅助对象加锁之后再去判断，所以上面的实现方式增加了额外的开销，损失了性能，为了改进上面实现方式的缺陷，我们只需要在**lock**语句前面加一句（**uniqueInstance==null**）的判断就可以避免锁所增加的额外开销，这种实现方式我们就叫它“双重锁定”，下面具体看看实现代码的：

```
/// <summary>
    /// 单例模式的实现
    /// </summary>
    public class Singleton
    {
        // 定义一个静态变量来保存类的实例
        private static Singleton uniqueInstance;

        // 定义一个标识确保线程同步
        private static readonly object locker = new object();

        // 定义私有构造函数，使外界不能创建该类实例
        private Singleton()
        {
        }

        /// <summary>
        /// 定义公有方法提供一个全局访问点,同时你也可以定义公有属性来提供全局
        /// </summary>
        /// <returns></returns>
        public static Singleton GetInstance()
        {
            // 当第一个线程运行到这里时，此时会对locker对象 "加锁",
            // 当第二个线程运行该方法时，首先检测到locker对象为"加锁"状态,
            // lock语句运行完之后（即线程运行完之后）会对该对象"解锁"
            // 双重锁定只需要一句判断就可以了
            if (uniqueInstance == null)
            {
                lock (locker)
                {
                    // 如果类的实例不存在则创建，否则直接返回
                    if (uniqueInstance == null)
                    {
                        uniqueInstance = new Singleton();
                    }
                }
            }
            return uniqueInstance;
        }
    }
```

五、C#中实现了单例模式的类

理解完了单例模式之后，菜鸟又接着问了：.NET FrameWork类库中有没有单例模式的实现呢？

经过查看，.NET类库中确实存在单例模式的实现类，不过该类不是公开的，下面就具体看看该类的一个实现的(该类具体存在于System.dll程序集，命名空间为System,大家可以用反射工具Reflector去查看源码的):

```
// 该类不是一个公开类
    // 但是该类的实现应用了单例模式
    internal sealed class SR
    {
        private static SR loader;
        internal SR()
        {
        }
        // 主要是因为该类不是公有，所以这个全部访问点也定义为私有的了
        // 但是思想还是用到了单例模式的思想的
        private static SR GetLoader()
        {
            if (loader == null)
            {
                SR sr = new SR();
                Interlocked.CompareExchange<SR>(ref loader, sr, nul
            }
            return loader;
        }

        // 这个公有方法中调用了GetLoader方法的
        public static object GetObject(string name)
        {
            SR loader = GetLoader();
            if (loader == null)
            {
                return null;
            }
            return loader.resources.GetObject(name, Culture);
        }
    }
```

六、总结

到这里，设计模式的单例模式就介绍完了，希望通过本文章大家可以对单例模式有一个更深的理解，并且希望之前没接触过单例模式或觉得单例模式陌生的朋友看完之后会惊叹：原来如此！

C#设计模式(2)——简单工厂模式

一、引言

这个系列也是自己对设计模式的一些学习笔记,希望对一些初学设计模式的人有所帮助的,在上一个专题中介绍了单例模式,在这个专题中继续为大家介绍一个比较容易理解的模式——简单工厂模式。

二、简单工厂模式的介绍

说到简单工厂，自然的第一个疑问当然就是什么是简单工厂模式了？在现实生活中工厂是负责生产产品的,同样在设计模式中,简单工厂模式我们也可以理解为负责生产对象的一个类,我们平常编程中，当使用"new"关键字创建一个对象时，此时该类就依赖与这个对象，也就是他们之间的耦合度高，当需求变化时，我们就不得不去修改此类的源码，此时我们可以运用面向对象（OO）的很重要的原则去解决这一的问题，该原则就是——封装改变，既然要封装改变，自然也就要找到改变的代码，然后把改变的代码用类来封装，这样的一种思路也就是我们简单工厂模式的实现方式了。下面通过一个现实生活中的例子来引出简单工厂模式。

在外面打工的人，免不了要经常在外面吃饭，当然我们也可以自己在家做饭吃，但是自己做饭吃麻烦，因为又要自己买菜，然而，出去吃饭就完全没有这些麻烦的，我们只需要到餐馆点菜就可以了，买菜的事情就交给餐馆做就可以了，这里餐馆就充当简单工厂的角色，下面让我们看看现实生活中的例子用代码是怎样来表现的。

自己做饭的情况：

```
/// <summary>
    /// 自己做饭的情况
    /// 没有简单工厂之前，客户想吃什么菜只能自己炒的
    /// </summary>
    public class Customer
    {
        /// <summary>
        /// 烧菜方法
        /// </summary>
        /// <param name="type"></param>
        /// <returns></returns>
        public static Food Cook(string type)
        {
            Food food = null;
            // 客户A说：我想吃西红柿炒蛋怎么办？
            // 客户B说：那你就自己烧啊
            // 客户A说： 好吧，那就自己做吧
            if (type.Equals("西红柿炒蛋"))
            {
```

```
                food = new TomatoScrambledEggs();
            }
            // 我又想吃土豆肉丝, 这个还是得自己做
            // 我觉得自己做好累哦，如果能有人帮我做就好了？
            else if (type.Equals("土豆肉丝"))
            {
                food = new ShreddedPorkWithPotatoes();
            }
            return food;
        }

        static void Main(string[] args)
        {
            // 做西红柿炒蛋
            Food food1 = Cook("西红柿炒蛋");
            food1.Print();

            Food food2 = Cook("土豆肉丝");
            food2.Print();

            Console.Read();
        }
    }
 /// <summary>
    /// 菜抽象类
    /// </summary>
    public abstract class Food
    {
        // 输出点了什么菜
        public abstract void Print();
    }

    /// <summary>
    /// 西红柿炒鸡蛋这道菜
    /// </summary>
    public class TomatoScrambledEggs : Food
    {
        public override void Print()
        {
            Console.WriteLine("一份西红柿炒蛋！");
        }
    }

    /// <summary>
    /// 土豆肉丝这道菜
    /// </summary>
    public class ShreddedPorkWithPotatoes : Food
    {
        public override void Print()
        {
            Console.WriteLine("一份土豆肉丝");
        }
    }
```

自己做饭，如果我们想吃别的菜时，此时就需要去买这种菜和洗菜这些繁琐的操作，有了餐馆（也就是简单工厂）之后，我们就可以把这些操作交给餐馆去做，此时消费者（也就是我们）对菜（也就是具体对象）的依赖关系从直接变成的间接的，这样就是实现了面向对象的另一个原则——降低对象之间的耦合度，下面就具体看看有了餐馆之后的实现代码（即简单工厂的实现）：

```
/// <summary>
    /// 顾客充当客户端，负责调用简单工厂来生产对象
    /// 即客户点菜，厨师（相当于简单工厂）负责烧菜(生产的对象)
    /// </summary>
    class Customer
    {
        static void Main(string[] args)
        {
            // 客户想点一个西红柿炒蛋
            Food food1 = FoodSimpleFactory.CreateFood("西红柿炒蛋");
            food1.Print();

            // 客户想点一个土豆肉丝
            Food food2 = FoodSimpleFactory.CreateFood("土豆肉丝");
            food2.Print();

            Console.Read();
        }
    }

    /// <summary>
    /// 菜抽象类
    /// </summary>
    public abstract class Food
    {
        // 输出点了什么菜
        public abstract void Print();
    }

    /// <summary>
    /// 西红柿炒鸡蛋这道菜
    /// </summary>
    public class TomatoScrambledEggs : Food
    {
        public override void Print()
        {
            Console.WriteLine("一份西红柿炒蛋！");
        }
    }

    /// <summary>
    /// 土豆肉丝这道菜
    /// </summary>
    public class ShreddedPorkWithPotatoes : Food
```

```
    {
        public override void Print()
        {
            Console.WriteLine("一份土豆肉丝");
        }
    }

    /// <summary>
    /// 简单工厂类, 负责 炒菜
    /// </summary>
    public class FoodSimpleFactory
    {
        public static Food CreateFood(string type)
        {
            Food food = null;
            if (type.Equals("土豆肉丝"))
            {
                food= new ShreddedPorkWithPotatoes();
            }
            else if (type.Equals("西红柿炒蛋"))
            {
                food= new TomatoScrambledEggs();
            }

            return food;
        }
    }
```

三、优点与缺点

看完简单工厂模式的实现之后，你和你的小伙伴们肯定会有这样的疑惑（因为我学习的时候也有）——这样我们只是把变化移到了工厂类中罢了，好像没有变化的问题，因为如果客户想吃其他菜时，此时我们还是需要修改工厂类中的方法（也就是多加case语句，没应用简单工厂模式之前，修改的是客户类）。我首先要说：你和你的小伙伴很对，这个就是简单工厂模式的缺点所在（这个缺点后面介绍的工厂方法可以很好地解决），然而，简单工厂模式与之前的实现也有它的优点：

- 简单工厂模式解决了客户端直接依赖于具体对象的问题，客户端可以消除直接创建对象的责任，而仅仅是消费产品。简单工厂模式实现了对责任的分割。
- 简单工厂模式也起到了代码复用的作用，因为之前的实现（自己做饭的情况）中，换了一个人同样要去在自己的类中实现做菜的方法，然后有了简单工厂之后，去餐馆吃饭的所有人都不用那么麻烦了，只需要负责消费就可以了。此时简单工厂的烧菜方法就让所有客户共用了。（同时这点也是简单工厂方法的缺点——因为工厂类集中了所有产品创建逻辑，一旦不能正常工作，整个系统都会受到影响，也没什么不好理解的，就如事物都有两面性一样道理）

虽然上面已经介绍了简单工厂模式的缺点，下面还是总结下简单工厂模式的缺点：

- 工厂类集中了所有产品创建逻辑，一旦不能正常工作，整个系统都会受到影响

（通俗地意思就是：一旦餐馆没饭或者关门了，很多不愿意做饭的人就没饭吃了）

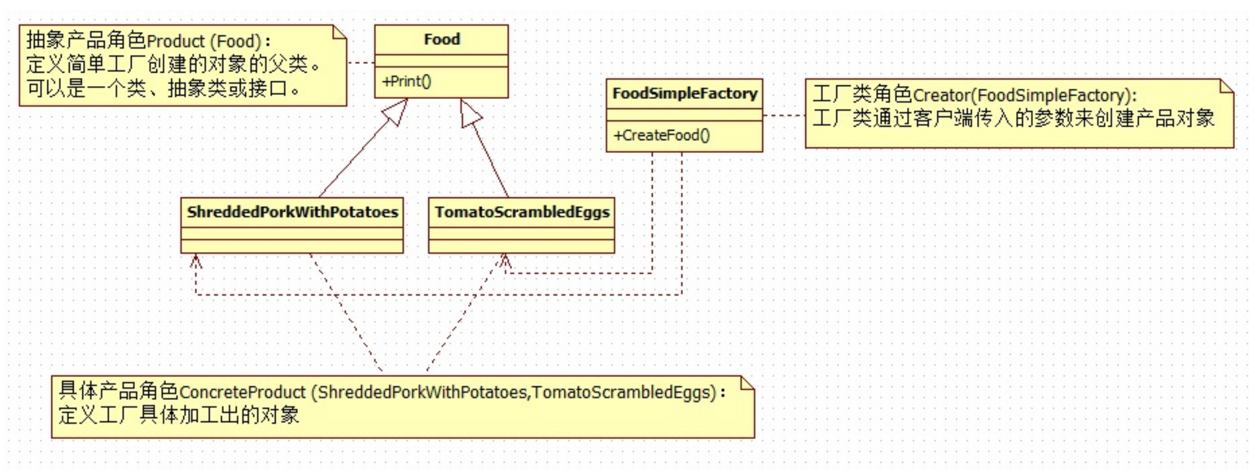
- 系统扩展困难，一旦添加新产品就不得不修改工厂逻辑，这样就会造成工厂逻辑过于复杂。

了解了简单工厂模式之后的优缺点之后，我们之后就可以知道简单工厂的应用场景了：

- 当工厂类负责创建的对象比较少时可以考虑使用简单工厂模式（）
- 客户如果只知道传入工厂类的参数，对于如何创建对象的逻辑不关心时可以考虑使用简单工厂模式

四、简单工厂模式UML图

简单工厂模式又叫静态方法模式（因为工厂类都定义了一个静态方法），由一个工厂类根据传入的参数决定创建出哪一种产品类的实例（通俗点表达：通过客户下的订单来负责烧那种菜）。简单工厂模式的UML图如下：



五、.NET中简单工厂模式的实现

介绍完了简单工厂模式之后，我学习的时候就像：.NET类库中是否有实现了简单工厂模式的类呢？后面确实有，.NET中System.Text.Encoding类就实现了简单工厂模式，该类中的**GetEncoding(int codepage)**就是工厂方法，具体的代码可以通过Reflector反编译工具进行查看，下面我也贴出该方法中部分代码：

```
public static Encoding GetEncoding(int codepage)
{
    Encoding unicode = null;
    if (encodings != null)
    {
        unicode = (Encoding) encodings[codepage];
    }
    if (unicode == null)
    {
        object obj2;
```

```
            bool lockTaken = false;
            try
            {
                Monitor.Enter(obj2 = InternalSyncObject, ref lockTaken)
                if (encodings == null)
                {
                    encodings = new Hashtable();
                }
                unicode = (Encoding) encodings[codepage];
                if (unicode != null)
                {
                    return unicode;
                }
                switch (codepage)
                {
                    case 0:
                        unicode = Default;
                        break;

                    case 1:
                    case 2:
                    case 3:
                    case 0x2a:
                        throw new ArgumentException(Environment.GetReso

                    case 0x4b0:
                        unicode = Unicode;
                        break;

                    case 0x4b1:
                        unicode = BigEndianUnicode;
                        break;

                    case 0x6faf:
                        unicode = Latin1;
                        break;

                    case 0xfde9:
                        unicode = UTF8;
                        break;

                    case 0x4e4:
                        unicode = new SBCSCodePageEncoding(codepage);
                        break;

                    case 0x4e9f:
                        unicode = ASCII;
                        break;

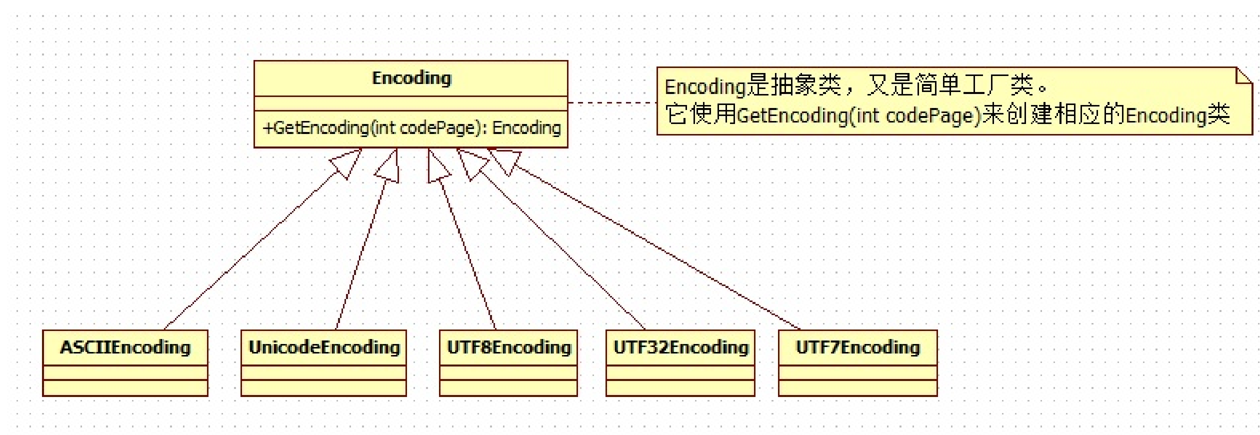
                    default:
                        unicode = GetEncodingCodePage(codepage);
                        if (unicode == null)
                        {
```

```
                    unicode = GetEncodingRare(codepage);
                }
                break;
            }
            encodings.Add(codepage, unicode);
            return unicode;

        }
}
```

View Code

.NET 中Encoding的UML图为：



Encoding类中实现的简单工厂模式是简单工厂模式的一种演变，此时简单工厂类由抽象产品角色扮演，然而.NET中Encoding类是如何解决简单工厂模式中存在的问题的呢（即如果新添加一种编码怎么办）？在GetEncoding方法里的switch函数有如下代码：

```
switch (codepage)
    {
        .......
  default:
                    unicode = GetEncodingCodePage(codepage);
                    if (unicode == null)
                    {
                        unicode = **GetEncodingRare**(codepage); //
                    }
                    break;
            ......
     }
```

在GetEncodingRare方法里有一些不常用编码的实例化代码，微软正式通过这个方法来解决新增加一种编码的问题。（其实也就是列出所有可能的编码情况），微软之所以以这样的方式来解决这个问题，可能是由于现在编码已经稳定了，添加新编码的可能性比较低，所以在.NET 4.5仍然未改动这部分代码。

六、总结

到这里，简单工厂模式的介绍都到这里了，后面将介绍工厂方法模式来解决简单工厂模式中存在的问题。

本专题中的全部源码：简单工厂模式源码

C#设计模式(3)——工厂方法模式

一、引言

在简单工厂模式中讲到简单工厂模式的缺点，有一点是——简单工厂模式系统难以扩展，一旦添加新产品就不得不修改简单工厂方法，这样就会造成简单工厂的实现逻辑过于复杂，然而本专题介绍的工厂方法模式可以解决简单工厂模式中存在的这个问题，下面就具体看看工厂模式是如何解决该问题的。

二、工厂方法模式的实现

工厂方法模式之所以可以解决简单工厂的模式，是因为它的实现把具体产品的创建推迟到子类中，此时工厂类不再负责所有产品的创建，而只是给出具体工厂必须实现的接口，这样工厂方法模式就可以允许系统不修改工厂类逻辑的情况下来添加新产品，这样也就克服了简单工厂模式中缺点。下面看下工厂模式的具体实现代码（这里还是以简单工厂模式中点菜的例子来实现）：

```
namespace 设计模式之工厂方法模式
{
    /// <summary>
    /// 菜抽象类
    /// </summary>
    public abstract class Food
    {
        // 输出点了什么菜
        public abstract void Print();
    }

    /// <summary>
    /// 西红柿炒鸡蛋这道菜
    /// </summary>
    public class TomatoScrambledEggs : Food
    {
        public override void Print()
        {
            Console.WriteLine("西红柿炒蛋好了！");
        }
    }

    /// <summary>
    /// 土豆肉丝这道菜
    /// </summary>
    public class ShreddedPorkWithPotatoes : Food
    {
        public override void Print()
        {
```

```
            Console.WriteLine("土豆肉丝好了");
        }
    }

    /// <summary>
    /// 抽象工厂类
    /// </summary>
    public abstract class Creator
    {
        // 工厂方法
        public abstract Food CreateFoddFactory();
    }

    /// <summary>
    /// 西红柿炒蛋工厂类
    /// </summary>
    public class TomatoScrambledEggsFactory:Creator
    {
        /// <summary>
        /// 负责创建西红柿炒蛋这道菜
        /// </summary>
        /// <returns></returns>
        public override Food CreateFoddFactory()
        {
            return new TomatoScrambledEggs();
        }
    }

    /// <summary>
    /// 土豆肉丝工厂类
    /// </summary>
    public class ShreddedPorkWithPotatoesFactory:Creator
    {
        /// <summary>
        /// 负责创建土豆肉丝这道菜
        /// </summary>
        /// <returns></returns>
        public override Food CreateFoddFactory()
        {
            return new ShreddedPorkWithPotatoes();
        }
    }

    /// <summary>
    /// 客户端调用
    /// </summary>
    class Client
    {
        static void Main(string[] args)
        {
            // 初始化做菜的两个工厂（）
            Creator shreddedPorkWithPotatoesFactory = new ShreddedP
            Creator tomatoScrambledEggsFactory = new TomatoScramble
```

```
                // 开始做西红柿炒蛋
                Food tomatoScrambleEggs = tomatoScrambledEggsFactory.Cr
                tomatoScrambleEggs.Print();

                //开始做土豆肉丝
                Food shreddedPorkWithPotatoes = shreddedPorkWithPotatoe
                shreddedPorkWithPotatoes.Print();

                Console.Read();
            }
        }
    }
```

使用工厂方法实现的系统，如果系统需要添加新产品时，我们可以利用多态性来完成系统的扩展，对于抽象工厂类和具体工厂中的代码都不需要做任何改动。例如，我们我们还想点一个“肉末茄子”，此时我们只需要定义一个肉末茄子具体工厂类和肉末茄子类就可以。而不用像简单工厂模式中那样去修改工厂类中的实现（具体指添加case语句)。具体代码为：

```
/// <summary>
    /// 肉末茄子这道菜
    /// </summary>
    public class MincedMeatEggplant : Food
    {
        /// <summary>
        /// 重写抽象类中的方法
        /// </summary>
        public override void Print()
        {
            Console.WriteLine("肉末茄子好了");
        }
    }
/// <summary>
    /// 肉末茄子工厂类，负责创建肉末茄子这道菜
    /// </summary>
    public class MincedMeatEggplantFactory : Creator
    {
        /// <summary>
        /// 负责创建肉末茄子这道菜
        /// </summary>
        /// <returns></returns>
        public override Food CreateFoddFactory()
        {
            return new MincedMeatEggplant();
        }
    }

    /// <summary>
    /// 客户端调用
    /// </summary>
    class Client
    {
        static void Main(string[] args)
        {

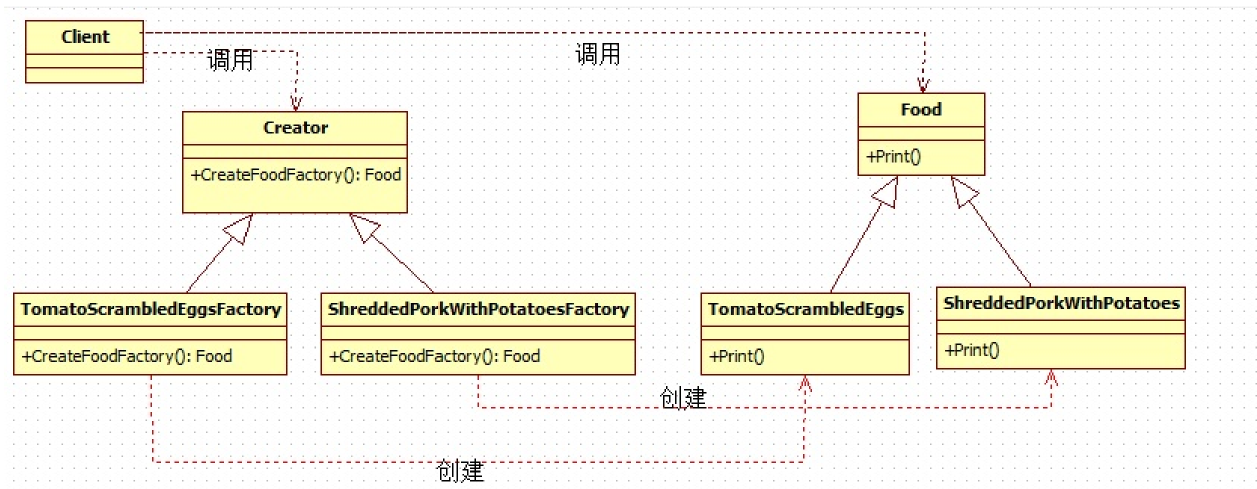
            // 如果客户又想点肉末茄子了
            // 再另外初始化一个肉末茄子工厂
            Creator minceMeatEggplantFactor = new MincedMeatEggplar

            // 利用肉末茄子工厂来创建肉末茄子这道菜
            Food minceMeatEggplant = minceMeatEggplantFactor.Create
            minceMeatEggplant.Print();

            Console.Read();
        }
    }
```

三、工厂方法模式的UML图

讲解完工厂模式的具体实现之后，让我们看下工厂模式中各类之间的UML图：



从UML图可以看出，在工厂方法模式中，工厂类与具体产品类具有平行的等级结构，它们之间是一一对应的。针对UML图的解释如下：

Creator类：充当抽象工厂角色，任何具体工厂都必须继承该抽象类

TomatoScrambledEggsFactory和ShreddedPorkWithPotatoesFactory类：充当具体工厂角色，用来创建具体产品

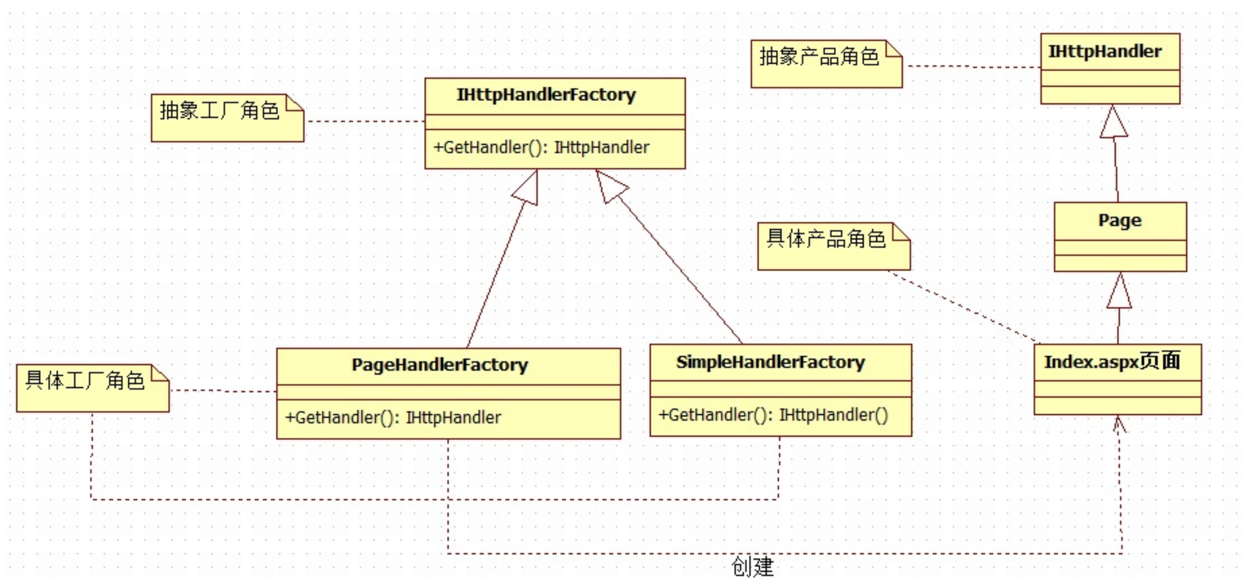
Food类：充当抽象产品角色，具体产品的抽象类。任何具体产品都应该继承该类

TomatoScrambledEggs和ShreddedPorkWithPotatoes类：充当具体产品角色，实现抽象产品类对定义的抽象方法，由具体工厂类创建，它们之间有一一对应的关系。

四、.NET中实现了工厂方法的类

.NET 类库中也有很多实现了工厂方法的类，例如Asp.net中，处理程序对象是具体用来处理请求，当我们请求一个*.aspx*的文件时，此时会映射到***System.Web.UI.PageHandlerFactory***类上进行处理，而对.ashx的请求将映射到**System.Web.UI.SimpleHandlerFactory**类中（这两个类都是继承于IHttpHandlerFactory接口的），关于这点说明我们可以在“C:\Windows\Microsoft.NET\Framework\v4.0.30319\Config\Web.Config”文件中找到相关定义，具体定义如下：

下面我们就具体看下工厂方法模式在Asp.net中是如何实现的，如果对一个Index.aspx页面发出请求时，将会调用**PageHandlerFactory**中GetHandler方法来创建一个Index.aspx对象，它们之间的类图关系如下：



五、总结

工厂方法模式通过面向对象编程中的多态性来将对象的创建延迟到具体工厂中，从而解决了简单工厂模式中存在的问题，也很好地符合了开放封闭原则（即对扩展开发，对修改封闭）。

C#设计模式(4)——抽象工厂模式

一、引言

在上一专题中介绍了工厂方法模式，工厂方法模式是为了克服简单工厂模式的缺点而设计出来的,简单工厂模式的工厂类随着产品类的增加需要增加额外的代码），而工厂方法模式每个具体工厂类只完成单个实例的创建,所以它具有很好的可扩展性。但是在现实生活中，一个工厂只创建单个产品这样的例子很少，因为现在的工厂都多元化了，一个工厂创建一系列的产品，如果我们要设计这样的系统时，工厂方法模式显然在这里不适用，然后抽象工厂模式却可以很好地解决一系列产品创建的问题,这是本专题所要介绍的内容。

二、抽象工厂详细介绍

这里首先以一个生活中抽象工厂的例子来实现一个抽象工厂，然后再给出抽象工厂的定义和UML图来帮助大家更好地掌握抽象工厂模式，同时大家在理解的时候，可以对照抽象工厂生活中例子的实现和它的定义来加深抽象工厂的UML图理解。

2.1 抽象工厂的具体实现

下面就以生活中 “绝味” 连锁店的例子来实现一个抽象工厂模式。例如，绝味鸭脖想在江西南昌和上海开分店，但是由于当地人的口味不一样，在南昌的所有绝味的东西会做的辣一点，而上海不喜欢吃辣的，所以上海的所有绝味的东西都不会做的像南昌的那样辣，然而这点不同导致南昌绝味工厂和上海的绝味工厂生成所有绝味的产品都不同，也就是某个具体工厂需要负责一系列产品(指的是绝味所有食物)的创建工作，下面就具体看看如何使用抽象工厂模式来实现这种情况。

```
 1 /// <summary>
 2     /// 下面以绝味鸭脖连锁店为例子演示下抽象工厂模式
 3     /// 因为每个地方的喜欢的口味不一样，有些地方喜欢辣点的，有些地方喜欢吃
 4     /// 客户端调用
 5     /// </summary>
 6     class Client
 7     {
 8         static void Main(string[] args)
 9         {
10             // 南昌工厂制作南昌的鸭脖和鸭架
11             AbstractFactory nanChangFactory = new NanChangFacto
12             YaBo nanChangYabo = nanChangFactory.CreateYaBo();
13             nanChangYabo.Print();
14             YaJia nanChangYajia= nanChangFactory.CreateYaJia();
15             nanChangYajia.Print();
16 
17             // 上海工厂制作上海的鸭脖和鸭架
```

```
18              AbstractFactory shangHaiFactory = new ShangHaiFacto
19              shangHaiFactory.CreateYaBo().Print();
20              shangHaiFactory.CreateYaJia().Print();
21
22              Console.Read();
23          }
24      }
25
26      /// <summary>
27      /// 抽象工厂类，提供创建两个不同地方的鸭架和鸭脖的接口
28      /// </summary>
29      public abstract class AbstractFactory
30      {
31          // 抽象工厂提供创建一系列产品的接口，这里作为例子，只给出了绝味中
32          public abstract YaBo CreateYaBo();
33          public abstract YaJia CreateYaJia();
34      }
35
36      /// <summary>
37      /// 南昌绝味工厂负责制作南昌的鸭脖和鸭架
38      /// </summary>
39      public class NanChangFactory : AbstractFactory
40      {
41          // 制作南昌鸭脖
42          public override YaBo CreateYaBo()
43          {
44              return new NanChangYaBo();
45          }
46          // 制作南昌鸭架
47          public override YaJia CreateYaJia()
48          {
49              return new NanChangYaJia();
50          }
51      }
52
53      /// <summary>
54      /// 上海绝味工厂负责制作上海的鸭脖和鸭架
55      /// </summary>
56      public class ShangHaiFactory : AbstractFactory
57      {
58          // 制作上海鸭脖
59          public override YaBo CreateYaBo()
60          {
61              return new ShangHaiYaBo();
62          }
63          // 制作上海鸭架
64          public override YaJia CreateYaJia()
65          {
66              return new ShangHaiYaJia();
67          }
68      }
69
70      /// <summary>
```

```
 71     /// 鸭脖抽象类，供每个地方的鸭脖类继承
 72     /// </summary>
 73     public abstract class YaBo
 74     {
 75         /// <summary>
 76         /// 打印方法，用于输出信息
 77         /// </summary>
 78         public abstract void Print();
 79     }
 80 
 81     /// <summary>
 82     /// 鸭架抽象类，供每个地方的鸭架类继承
 83     /// </summary>
 84     public abstract class YaJia
 85     {
 86         /// <summary>
 87         /// 打印方法，用于输出信息
 88         /// </summary>
 89         public abstract void Print();
 90     }
 91 
 92     /// <summary>
 93     /// 南昌的鸭脖类，因为江西人喜欢吃辣的，所以南昌的鸭脖稍微会比上海做的
 94     /// </summary>
 95     public class NanChangYaBo : YaBo
 96     {
 97         public override void Print()
 98         {
 99             Console.WriteLine("南昌的鸭脖");
100         }
101     }
102 
103     /// <summary>
104     /// 上海的鸭脖没有南昌的鸭脖做的辣
105     /// </summary>
106     public class ShangHaiYaBo : YaBo
107     {
108         public override void Print()
109         {
110             Console.WriteLine("上海的鸭脖");
111         }
112     }
113 
114     /// <summary>
115     /// 南昌的鸭架
116     /// </summary>
117     public class NanChangYaJia : YaJia
118     {
119         public override void Print()
120         {
121             Console.WriteLine("南昌的鸭架子");
122         }
123     }
```

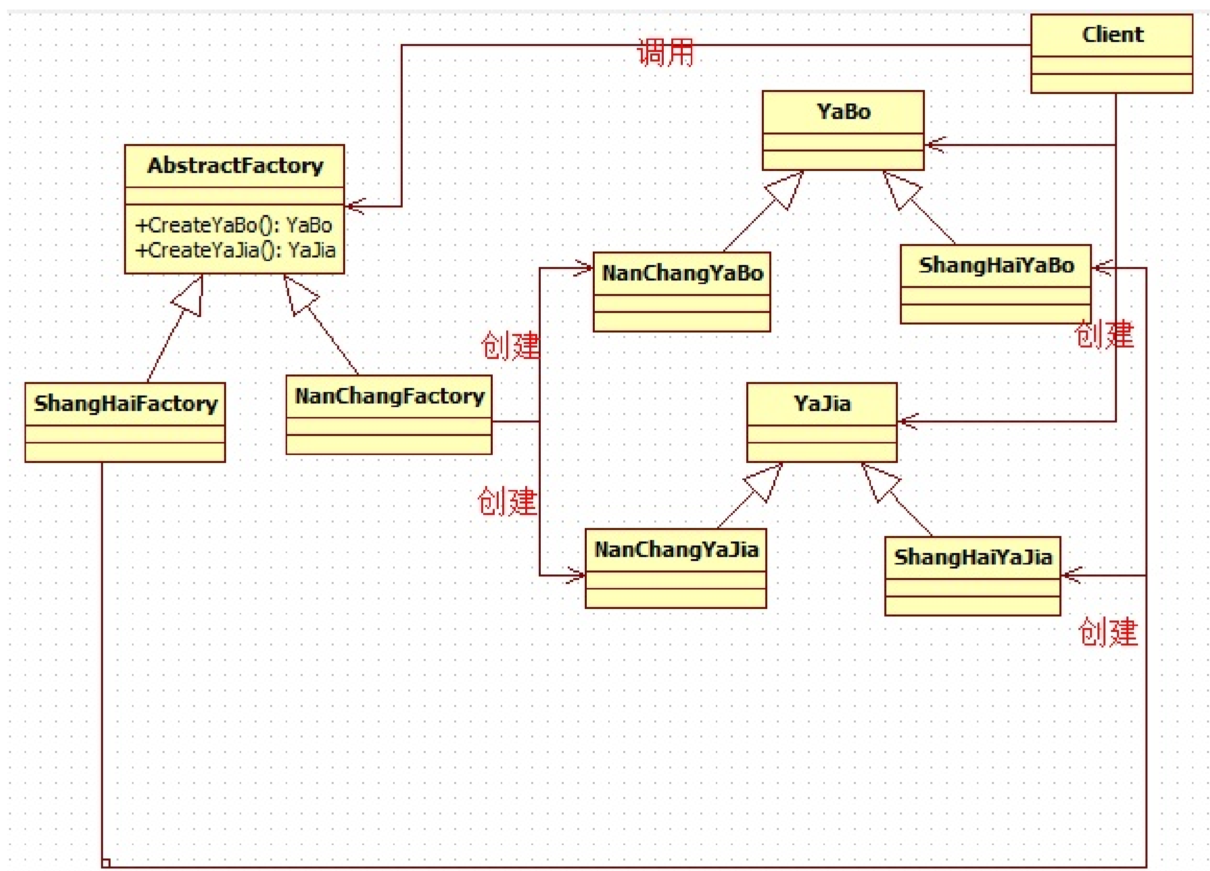
```
124
125     /// <summary>
126     /// 上海的鸭架
127     /// </summary>
128     public class ShangHaiYaJia : YaJia
129     {
130         public override void Print()
131         {
132             Console.WriteLine("上海的鸭架子");
133         }
134     }
```

2.2 抽象工厂模式的定义和类图

上面代码中都有详细的注释，这里就不再解释上面的代码了，下面就具体看看抽象工厂模式的定义吧（理解定义可以参考上面的实现来加深理解）：

抽象工厂模式：提供一个创建产品的接口来负责创建相关或依赖的对象，而不具体明确指定具体类

抽象工厂允许客户使用抽象的接口来创建一组相关产品，而不需要知道或关心实际生产出的具体产品是什么。这样客户就可以从具体产品中被解耦。下面通过抽象工模式的类图来了解各个类中之间的关系：



2.3 抽象工厂应对需求变更

看完上面抽象工厂的实现之后，如果“绝味”公司又想在湖南开一家分店怎么办呢？因为湖南人喜欢吃麻辣的，下面就具体看看应用了抽象工厂模式的系统是如何应对这种需求的。

```
/// <summary>
    /// 如果绝味又想开一家湖南的分店时，因为湖南喜欢吃麻的
    /// 所以这是有需要有一家湖南的工厂专门制作
    /// </summary>
    public class HuNanFactory : AbstractFactory
    {
        // 制作湖南鸭脖
        public override YaBo CreateYaBo()
        {
            return new HuNanYaBo();
        }

        // 制作湖南鸭架
        public override YaJia CreateYaJia()
        {
            return new HuNanYajia();
        }
    }

    /// <summary>
    /// 湖南的鸭脖
    /// </summary>
    public class HuNanYaBo : YaBo
    {
        public override void Print()
        {
            Console.WriteLine("湖南的鸭脖");
        }
    }

    /// <summary>
    /// 湖南的鸭架
    /// </summary>
    public class HuNanYajia : YaJia
    {
        public override void Print()
        {
            Console.WriteLine("湖南的鸭架子");
        }
    }
```

此时，只需要添加三个类：一个是湖南具体工厂类，负责创建湖南口味的鸭脖和鸭架，另外两个类是具有湖南口味的鸭脖类和鸭架类。从上面代码看出，抽象工厂对于系列产品的变化支持“开放——封闭”原则（指的是要求系统对扩展开放，对修改封闭），扩展起来非常简便，但是，抽象工厂对于添加新产品这种情况就不支持”开放——封闭“原则，这也是抽象工厂的缺点所在，这点会在第四部分详细介绍。

三、抽象工厂的分析

抽象工厂模式将具体产品的创建延迟到具体工厂的子类中，这样将对象的创建封装起来，可以减少客户端与具体产品类之间的依赖，从而使系统耦合度低，这样更有利于后期的维护和扩展，这真是抽象工厂模式的优点所在，然后抽象模式同时也存在不足的地方。下面就具体看下抽象工厂的缺点（缺点其实在前面的介绍中以已经涉及了）：

抽象工厂模式很难支持新种类产品的变化。这是因为抽象工厂接口中已经确定了可以被创建的产品集合，如果需要添加新产品，此时就必须去修改抽象工厂的接口，这样就涉及到抽象工厂类的以及所有子类的改变，这样也就违背了“开发——封闭”原则。

知道了抽象工厂的优缺点之后，也就能很好地把握什么情况下考虑使用抽象工厂模式了，下面就具体看看使用抽象工厂模式的系统应该符合那几个前提：

- 一个系统不要求依赖产品类实例如何被创建、组合和表达的表达，这点也是所有工厂模式应用的前提。
- 这个系统有多个系列产品，而系统中只消费其中某一系列产品
- 系统要求提供一个产品类的库，所有产品以同样的接口出现，客户端不需要依赖具体实现。

四、.NET中抽象工厂模式实现

抽象工厂模式在实际中的应用也是相当频繁的，然而在我们.NET类库中也存在应用抽象工厂模式的类，这个类就是**System.Data.Common.DbProviderFactory**，这个类位于System.Data.dll程序集中,该类扮演抽象工厂模式中抽象工厂的角色，我们可以用reflector反编译工具查看该类的实现：

```
/// 扮演抽象工厂的角色
/// 创建连接数据库时所需要的对象集合,
/// 这个对象集合包括有 DbConnection对象（这个是抽象产品类,如绝味例子中的YaBo
public abstract class DbProviderFactory
{
    // 提供了创建具体产品的接口方法
    protected DbProviderFactory();
    public virtual DbCommand CreateCommand();
    public virtual DbCommandBuilder CreateCommandBuilder();
    public virtual DbConnection CreateConnection();
    public virtual DbConnectionStringBuilder CreateConnectionString
    public virtual DbDataAdapter CreateDataAdapter();
    public virtual DbDataSourceEnumerator CreateDataSourceEnumerato
    public virtual DbParameter CreateParameter();
    public virtual CodeAccessPermission CreatePermission(Permission
}
```

DbProviderFactory类是一个抽象工厂类，该类提供了创建数据库连接时所需要的对象集合的接口，实际创建的工作在其子类工厂中进行，微软使用的是SQL Server数据库，因此提供了连接SQL Server数据的具体工厂实现，具体代码可以用反编译工具查看，具体代码如下：

```
/// 扮演着具体工厂的角色，用来创建连接SQL Server数据所需要的对象
public sealed class SqlClientFactory : DbProviderFactory, IServiceP
{
    // Fields
    public static readonly SqlClientFactory Instance = new SqlClien

   // 构造函数
    private SqlClientFactory()
    {
    }

   // 重写抽象工厂中的方法
    public override DbCommand CreateCommand()
    {  // 创建具体产品
        return new SqlCommand();
    }

    public override DbCommandBuilder CreateCommandBuilder()
    {
        return new SqlCommandBuilder();
    }

    public override DbConnection CreateConnection()
    {
        return new SqlConnection();
    }

    public override DbConnectionStringBuilder CreateConnectionStrin
    {
        return new SqlConnectionStringBuilder();
    }

    public override DbDataAdapter CreateDataAdapter()
    {
        return new SqlDataAdapter();
    }

    public override DbDataSourceEnumerator CreateDataSourceEnumerat
    {
        return SqlDataSourceEnumerator.Instance;
    }

    public override DbParameter CreateParameter()
    {
        return new SqlParameter();
    }
```

```
        public override CodeAccessPermission CreatePermission(Permissio
        {
            return new SqlClientPermission(state);
        }
    }
```

因为微软只给出了连接SQL Server的具体工厂的实现，我们也可以自定义连接Oracle、MySql的具体工厂的实现。

五、总结

到这里，抽象工厂模式的介绍就结束，在下一专题就将为大家介绍建造模式。

程序源码：抽象工厂模式实现

C#设计模式(5)——建造者模式（Builder Pattern）

一、引言

在软件系统中，有时需要创建一个复杂对象，并且这个复杂对象由其各部分子对象通过一定的步骤组合而成。例如一个采购系统中，如果需要采购员去采购一批电脑时，在这个实际需求中，电脑就是一个复杂的对象，它是由CPU、主板、硬盘、显卡、机箱等组装而成的，如果此时让采购员一台一台电脑去组装的话真是要累死采购员了，这里就可以采用建造者模式来解决这个问题，我们可以把电脑的各个组件的组装过程封装到一个建造者类对象里，建造者只要负责返还给客户端全部组件都建造完毕的产品对象就可以了。然而现实生活中也是如此的，如果公司要采购一批电脑，此时采购员不可能自己去买各个组件并把它们组织起来，此时采购员只需要像电脑城的老板说自己要采购什么样的电脑就可以了，电脑城老板自然会把组装好的电脑送到公司。下面就以这个例子来展开建造者模式的介绍。

二、建造者模式的详细介绍

2.1 建筑者模式的具体实现

在这个例子中，电脑城的老板是直接与客户（也就是指采购员）联系的，然而电脑的组装是由老板指挥装机人员去把电脑的各个部件组装起来，真真负责创建产品（这里产品指的就是电脑）的人就是电脑城的装机人员。理清了这个逻辑过程之后，下面就具体看下如何用代码来表示这种现实生活中的逻辑过程：

```
 1 using System;
 2 using System.Collections.Generic;
 3 using System.Linq;
 4 using System.Text;
 5 
 6 
 7 /// <summary>
 8 /// 以组装电脑为例子
 9 /// 每台电脑的组成过程都是一致的，但是使用同样的构建过程可以创建不同的表示
10 /// 组装电脑的这个场景就可以应用建造者模式来设计
11 /// </summary>
12 namespace 设计模式之建造者模式
13 {
14     /// <summary>
15     /// 客户类
16     /// </summary>
17     class Customer
18     {
19         static void Main(string[] args)
```

```
20          {
21              // 客户找到电脑城老板说要买电脑，这里要装两台电脑
22              // 创建指挥者和构造者
23              Director director = new Director();
24              Builder b1 = new ConcreteBuilder1();
25              Builder b2 = new ConcreteBuilder2();
26
27              // 老板叫员工去组装第一台电脑
28              director.Construct(b1);
29
30              // 组装完，组装人员搬来组装好的电脑
31              Computer computer1 = b1.GetComputer();
32              computer1.Show();
33
34              // 老板叫员工去组装第二台电脑
35              director.Construct(b2);
36              Computer computer2 = b2.GetComputer();
37              computer2.Show();
38
39              Console.Read();
40          }
41      }
42
43      /// <summary>
44      /// 小王和小李难道会自愿地去组装嘛，谁不想休息的，这必须有一个人叫他们
45      /// 这个人当然就是老板了，也就是建造者模式中的指挥者
46      /// 指挥创建过程类
47      /// </summary>
48      public class Director
49      {
50          // 组装电脑
51          public void Construct(Builder builder)
52          {
53              builder.BuildPartCPU();
54              builder.BuildPartMainBoard();
55          }
56      }
57
58      /// <summary>
59      /// 电脑类
60      /// </summary>
61      public class Computer
62      {
63          // 电脑组件集合
64          private IList<string> parts = new List<string>();
65
66          // 把单个组件添加到电脑组件集合中
67          public void Add(string part)
68          {
69              parts.Add(part);
70          }
71
72          public void Show()
```

```
73          {
74              Console.WriteLine("电脑开始在组装.......");
75              foreach (string part in parts)
76              {
77                  Console.WriteLine("组件"+part+"已装好");
78              }
79
80              Console.WriteLine("电脑组装好了");
81          }
82      }
83
84      /// <summary>
85      /// 抽象建造者，这个场景下为 "组装人" ，这里也可以定义为接口
86      /// </summary>
87      public abstract class Builder
88      {
89          // 装CPU
90          public abstract void BuildPartCPU();
91          // 装主板
92          public abstract void BuildPartMainBoard();
93
94          // 当然还有装硬盘，电源等组件，这里省略
95
96          // 获得组装好的电脑
97          public abstract Computer GetComputer();
98      }
99
100     /// <summary>
101     /// 具体创建者，具体的某个人为具体创建者，例如：装机小王啊
102     /// </summary>
103     public class ConcreteBuilder1 : Builder
104     {
105         Computer computer = new Computer();
106         public override void BuildPartCPU()
107         {
108             computer.Add("CPU1");
109         }
110
111         public override void BuildPartMainBoard()
112         {
113             computer.Add("Main board1");
114         }
115
116         public override Computer GetComputer()
117         {
118             return computer;
119         }
120     }
121
122     /// <summary>
123     /// 具体创建者，具体的某个人为具体创建者，例如：装机小李啊
124     /// 又装另一台电脑了
125     /// </summary>
```

```
126      public class ConcreteBuilder2 : Builder
127      {
128          Computer computer = new Computer();
129          public override void BuildPartCPU()
130          {
131              computer.Add("CPU2");
132          }
133
134          public override void BuildPartMainBoard()
135          {
136              computer.Add("Main board2");
137          }
138
139          public override Computer GetComputer()
140          {
141              return computer;
142          }
143      }
144  }
```

上面代码中都有详细的注释代码，这里就不过多解释，大家可以参考代码和注释来与现实生活中的例子做对比，下图展示了上面代码的运行结果：

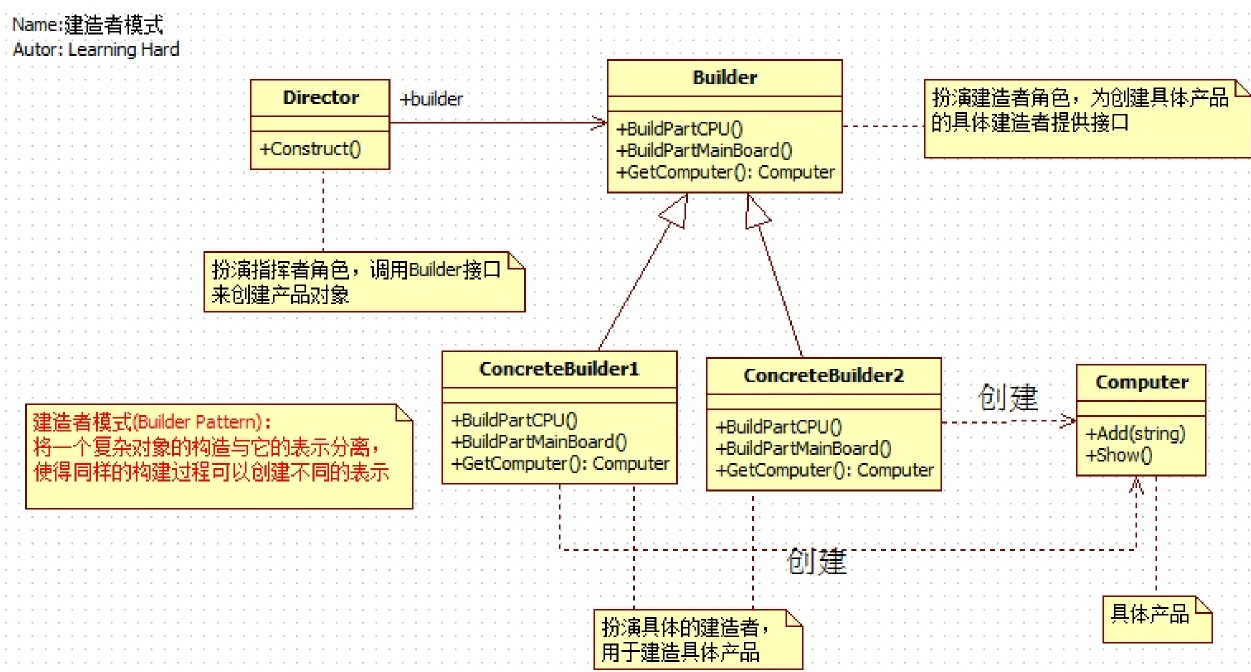
```
file:///F:/学习/博客园中例子/Projects/设计模式之建造者模式/设计模式之建造...
电脑开始在组装.......
组件CPU1已装好
组件Main board1已装好
电脑组装好了
电脑开始在组装.......
组件CPU2已装好
组件Main board2已装好
电脑组装好了
```

2.2 建造者模式的定义和类图

介绍完了建造者模式的具体实现之后，下面具体看下建造者模式的具体定义是怎样的。

建造者模式（Builder Pattern）:将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

建造者模式使得建造代码与表示代码的分离，可以使客户端不必知道产品内部组成的细节，从而降低了客户端与具体产品之间的耦合度，下面通过类图来帮助大家更好地理清建造者模式中类之间的关系。



三、建造者模式的分析

介绍完了建造者模式的具体实现之后，让我们总结下建造模式的实现要点：

1. 在建造者模式中，指挥者是直接与客户端打交道的，指挥者将客户端创建产品的请求划分为对各个部件的建造请求，再将这些请求委派到具体建造者角色，具体建造者角色是完成具体产品的构建工作的，却不为客户所知道。
2. 建造者模式主要用于“分步骤来构建一个复杂的对象”，其中“分步骤”是一个固定的组合过程，而复杂对象的各个部分是经常变化的（也就是说电脑的内部组件是经常变化的，这里指的的变化如硬盘的大小变了，CPU由单核变双核等）。
3. 产品不需要抽象类，由于建造模式的创建出来的最终产品可能差异很大，所以不大可能提炼出一个抽象产品类。
4. 在前面文章中介绍的抽象工厂模式解决了“系列产品”的需求变化，而建造者模式解决的是 “产品部分” 的需要变化。
5. 由于建造者隐藏了具体产品的组装过程，所以要改变一个产品的内部表示，只需要再实现一个具体的建造者就可以了，从而能很好地应对产品组成组件的需求变化。

四、.NET 中建造者模式的实现

前面的设计模式在.NET类库中都有相应的实现，那在.NET 类库中，是否也存在建造者模式的实现呢？ 然而对于疑问的答案是肯定的，在.NET 类库中，**System.Text.StringBuilder**(存在mscorlib.dll程序集中)就是一个建造者模式的实现。不过它的实现属于建造者模式的演化，此时的建造者模式没有指挥者角色和抽象建造者角色，**StringBuilder**类即扮演着具体建造者的角色，也同时扮演了指挥者和抽象建造者的角色，此时建造模式的实现如下：

```
/// <summary>
```

```
    /// 建造者模式的演变
    /// 省略了指挥者角色和抽象建造者角色
    /// 此时具体建造者角色扮演了指挥者和建造者两个角色
    /// </summary>
    public class Builder
    {
        // 具体建造者角色的代码
        private Product product = new Product();
        public void BuildPartA()
        {
            product.Add("PartA");
        }
        public void BuildPartB()
        {
            product.Add("PartB");
        }
        public Product GetProduct()
        {
            return product;
        }
        // 指挥者角色的代码
        public void Construct()
        {
            BuildPartA();
            BuildPartB();
        }
    }

    /// <summary>
    /// 产品类
    /// </summary>
    public class Product
    {
        // 产品组件集合
        private IList<string> parts = new List<string>();

        // 把单个组件添加到产品组件集合中
        public void Add(string part)
        {
            parts.Add(part);
        }

        public void Show()
        {
            Console.WriteLine("产品开始在组装.......");
            foreach (string part in parts)
            {
                Console.WriteLine("组件" + part + "已装好");
            }

            Console.WriteLine("产品组装完成");
        }
    }
```

```
// 此时客户端也要做相应调整
class Client
{
    private static Builder builder;
    static void Main(string[] args)
    {
        builder = new Builder();
        builder.Construct();
        Product product = builder.GetProduct();
        product.Show();
        Console.Read();
    }
}
```

StringBuilder类扮演着建造string对象的具体建造者角色，其中的ToString()方法用来返回具体产品给客户端（相当于上面代码中GetProduct方法）。其中Append方法用来创建产品的组件(相当于上面代码中BuildPartA和BuildPartB方法)，因为string对象中每个组件都是字符，所以也就不需要指挥者的角色的代码（指的是Construct方法,用来调用创建每个组件的方法来完成整个产品的组装），因为string字符串对象中每个组件都是一样的,都是字符,所以Append方法也充当了指挥者Construct方法的作用。

五、总结

到这里,建造者模式的介绍就结束了,建造者模式(Builder Pattern)，将一个复杂对象的构建与它的表示分离，使的同样的构建过程可以创建不同的表示。建造者模式的本质是使组装过程（用指挥者类进行封装，从而达到解耦的目的）和创建具体产品解耦,使我们不用去关心每个组件是如何组装的。

本专题中所有源码: 建造者模式源码

C#设计模式(6)——原型模式（Prototype Pattern）

一、引言

在软件系统中，当创建一个类的实例的过程很昂贵或很复杂，并且我们需要创建多个这样类的实例时，如果我们用new操作符去创建这样的类实例，这未免会增加创建类的复杂度和耗费更多的内存空间，因为这样在内存中分配了多个一样的类实例对象，然后如果采用工厂模式来创建这样的系统的话，随着产品类的不断增加，导致子类的数量不断增多，反而增加了系统复杂程度，所以在这里使用工厂模式来封装类创建过程并不合适，然而原型模式可以很好地解决这个问题，因为每个类实例都是相同的，当我们需要多个相同的类实例时，没必要每次都使用new运算符去创建相同的类实例对象，此时我们一般思路就是想——只创建一个类实例对象，如果后面需要更多这样的实例，可以通过对原来对象拷贝一份来完成创建，这样在内存中不需要创建多个相同的类实例，从而减少内存的消耗和达到类实例的复用。 然而这个思路正是原型模式的实现方式。下面就具体介绍下设计模式中的原型设计模式。

二、原型模式的详细介绍

在现实生活中，也有很多原型设计模式的例子，例如，细胞分裂的过程，一个细胞的有丝分裂产生两个相同的细胞；还有西游记中孙悟空变出后孙的本领和火影忍者中鸣人的隐分身忍术等。下面就以孙悟空为例子来演示下原型模式的实现。具体的实现代码如下：

```
///火影忍者中鸣人的影分身和孙悟空的的变都是原型模式
    class Client
    {
        static void Main(string[] args)
        {
            // 孙悟空 原型
            MonkeyKingPrototype prototypeMonkeyKing = new ConcreteP

            // 变一个
            MonkeyKingPrototype cloneMonkeyKing = prototypeMonkeyKi
            Console.WriteLine("Cloned1:\t"+cloneMonkeyKing.Id);

            // 变两个
            MonkeyKingPrototype cloneMonkeyKing2 = prototypeMonkeyK
            Console.WriteLine("Cloned2:\t" + cloneMonkeyKing2.Id);
            Console.ReadLine();
        }
    }
```

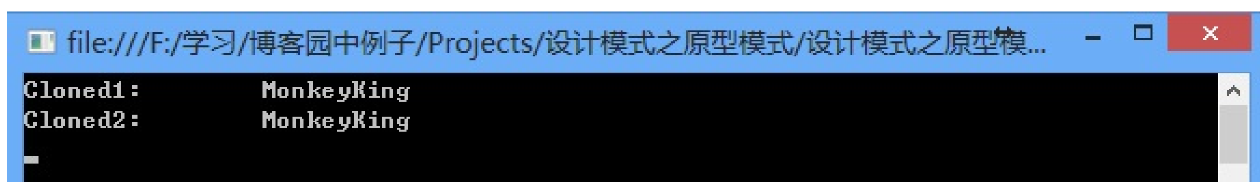
```
    /// <summary>
    /// 孙悟空原型
    /// </summary>
    public  abstract class MonkeyKingPrototype
    {
        public string Id { get; set; }
        public MonkeyKingPrototype(string id)
        {
            this.Id = id;
        }

        // 克隆方法，即孙大圣说“变”
        public abstract MonkeyKingPrototype Clone();
    }

    /// <summary>
    /// 创建具体原型
    /// </summary>
    public class ConcretePrototype : MonkeyKingPrototype
    {
        public ConcretePrototype(string id)
            : base(id)
        { }

        /// <summary>
        /// 浅拷贝
        /// </summary>
        /// <returns></returns>
        public override MonkeyKingPrototype Clone()
        {
            // 调用MemberwiseClone方法实现的是浅拷贝，另外还有深拷贝
            return (MonkeyKingPrototype)this.MemberwiseClone();
        }
    }
```

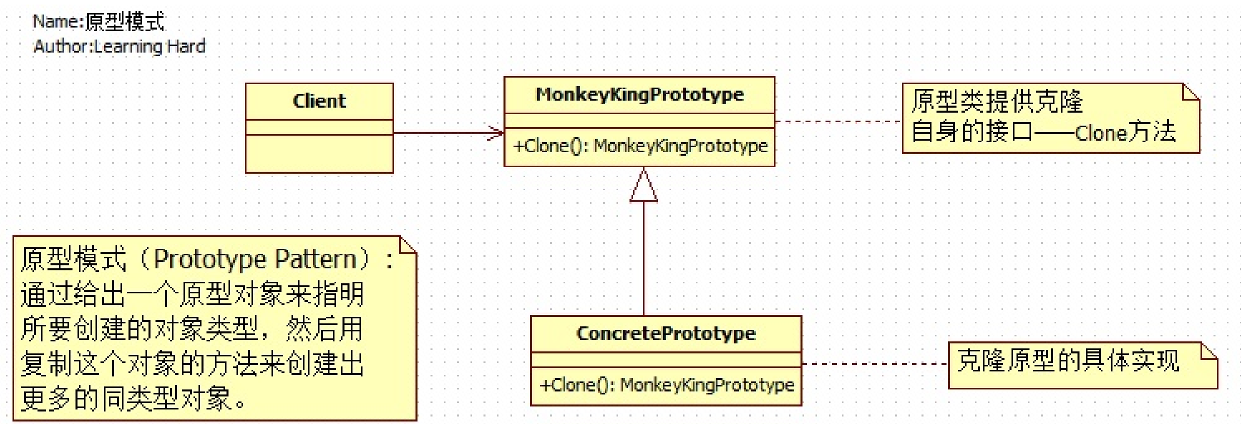
上面原型模式的运行结果为（从运行结果可以看出，创建的两个拷贝对象的ID属性都是与原型对象ID属性一样的）：



上面代码实现的浅拷贝的方式，浅拷贝是指当对象的字段值被拷贝时，字段引用的对象不会被拷贝。例如，如果一个对象有一个指向字符串的字段，并且我们对该对象做了一个浅拷贝，那么这两个对象将引用同一个字符串，而深拷贝是对对象实例中字段引用的对象也进行拷贝，如果一个对象有一个指向字符串的字段，并且我们对该对象进行了深拷贝的话，那么我们将创建一个对象和一个新的字符串，新的对

象将引用新的字符串。也就是说，执行深拷贝创建的新对象和原来对象不会共享任何东西，改变一个对象对另外一个对象没有任何影响，而执行浅拷贝创建的新对象与原来对象共享成员，改变一个对象，另外一个对象的成员也会改变。

介绍完原型模式的实现代码之后，下面看下原型模式的类图，通过类图来理清原型模式实现中类之间的关系。具体类图如下：



三、原型模式的优缺点

原型模式的优点有：

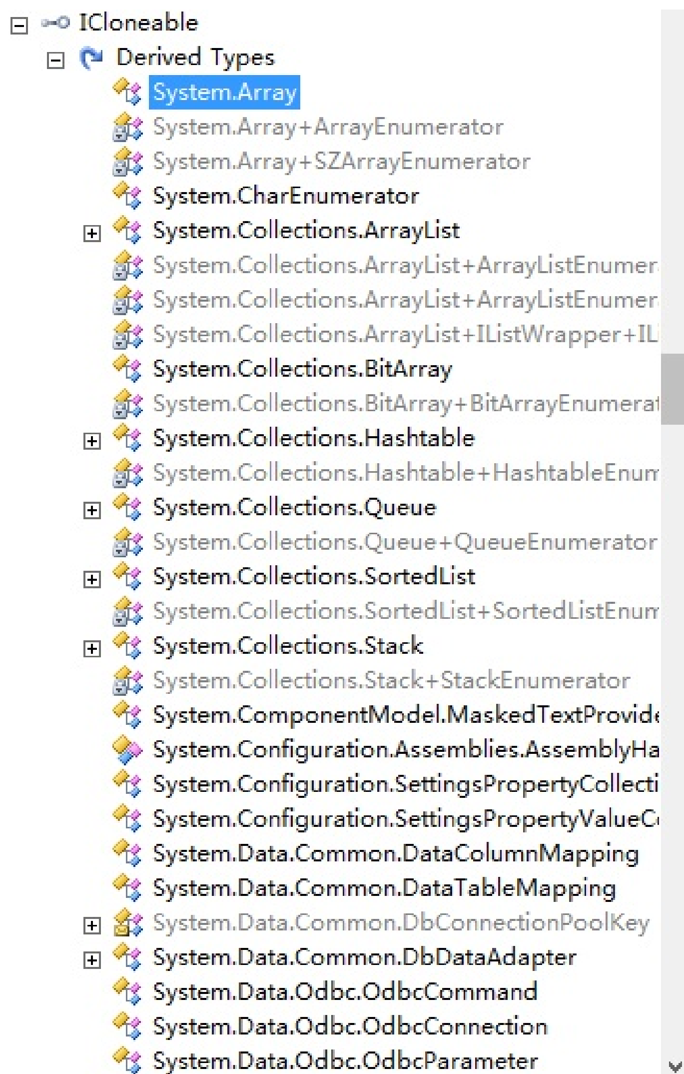
1. 原型模式向客户隐藏了创建新实例的复杂性
2. 原型模式允许动态增加或较少产品类。
3. 原型模式简化了实例的创建结构，工厂方法模式需要有一个与产品类等级结构相同的等级结构，而原型模式不需要这样。
4. 产品类不需要事先确定产品的等级结构，因为原型模式适用于任何的等级结构

原型模式的缺点有：

1. 每个类必须配备一个克隆方法
2. 配备克隆方法需要对类的功能进行通盘考虑，这对于全新的类不是很难，但对于已有的类不一定很容易，特别当一个类引用不支持串行化的间接对象，或者引用含有循环结构的时候。

四、.NET中原型模式的实现

在.NET中可以很容易地通过实现ICloneable接口（这个接口就是原型，提供克隆方法，相当于与上面代码中MonkeyKingPrototype抽象类）中Clone()方法来实现原型模式，如果我们想我们自定义的类具有克隆的功能，首先定义类继承与ICloneable接口并实现Clone方法。在.NET中实现了原型模式的类如下图所示（图中只截取了部分，可以用Reflector反编译工具进行查看）：



五、总结

到这里关于原型模式的介绍就结束了，原型模式用一个原型对象来指明所要创建的对象类型，然后用复制这个原型对象的方法来创建出更多的同类型对象，它与工厂方法模式的实现非常相似，其中原型模式中的Clone方法就类似工厂方法模式中的工厂方法，只是工厂方法模式的工厂方法是通过new运算符重新创建一个新的对象（相当于原型模式的深拷贝实现），而原型模式是通过调用MemberwiseClone方法来对原来对象进行拷贝，也就是复制，同时在原型模式优点中也介绍了与工厂方法的区别（第三点）。

本专题中所有源码：设计模式之原型模式

C#设计模式(7)——适配器模式（Adapter Pattern）

一、引言

在实际的开发过程中，由于应用环境的变化（例如使用语言的变化），我们需要的实现在新的环境中没有有现存对象可以满足，但是其他环境却存在这样现存的对象。那么如果将“将现存的对象”在新的环境中进行调用呢？解决这个问题的办法就是我们本文要介绍的适配器模式——使得新环境中不需要去重复实现已经存在了的实现而很好地把现有对象（指原来环境中的现有对象）加入到新环境来使用。

二、适配器模式的详细介绍

2.1 定义

下面让我们看看适配器的定义，适配器模式——把一个类的接口变换成客户端所期待的另一种接口，从而使原本接口不匹配而无法一起工作的两个类能够在一起工作。适配器模式有类的适配器模式和对象的适配器模式两种形式，下面我们分别讨论这两种形式的实现和给出对应的类图来帮助大家理清类之间的关系。

2.2 类的适配器模式实现

在这里以生活中的一个例子来进行演示适配器模式的实现，具体场景是: 在生活中，我们买的电器插头是2个孔的，但是我们买的插座只有三个孔的，此时我们就希望电器的插头可以转换为三个孔的就好，这样我们就可以直接把它插在插座上，此时三个孔插头就是客户端期待的另一种接口，自然两个孔的插头就是现有的接口，适配器模式就是用来完成这种转换的，具体实现代码如下：

```
using System;

/// 这里以插座和插头的例子来诠释适配器模式
/// 现在我们买的电器插头是2个孔，但是我们买的插座只有3个孔的
/// 这是我们想把电器插在插座上的话就需要一个电适配器
namespace 设计模式之适配器模式
{
    /// <summary>
    /// 客户端，客户想要把2个孔的插头 转变成三个孔的插头，这个转变交给适配器就
    /// 既然适配器需要完成这个功能，所以它必须同时具体2个孔插头和三个孔插头的特
    /// </summary>
    class Client
    {
        static void Main(string[] args)
        {
```

```
            // 现在客户端可以通过电适配要使用2个孔的插头了
            IThreeHole threehole = new PowerAdapter();
            threehole.Request();
            Console.ReadLine();
        }
    }

    /// <summary>
    /// 三个孔的插头，也就是适配器模式中的目标角色
    /// </summary>
    public interface IThreeHole
    {
        void Request();
    }

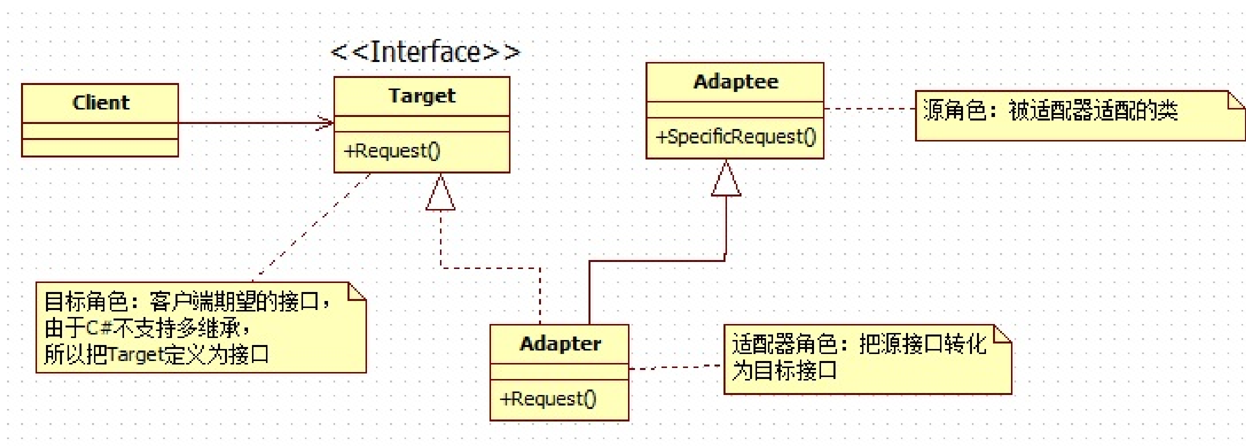
    /// <summary>
    /// 两个孔的插头，源角色——需要适配的类
    /// </summary>
    public abstract class TwoHole
    {
        public void SpecificRequest()
        {
            Console.WriteLine("我是两个孔的插头");
        }
    }

    /// <summary>
    /// 适配器类，接口要放在类的后面
    /// 适配器类提供了三个孔插头的行为，但其本质是调用两个孔插头的方法
    /// </summary>
    public class PowerAdapter:TwoHole,IThreeHole
    {
        /// <summary>
        /// 实现三个孔插头接口方法
        /// </summary>
        public void Request()
        {
            // 调用两个孔插头方法
            this.SpecificRequest();
        }
    }
}
```

从上面代码中可以看出，客户端希望调用Request方法（即三个孔插头），但是我们现有的类（即2个孔的插头）并没有Request方法，它只有SpecificRequest方法（即两个孔插头本身的方法），然而适配器类（适配器必须实现三个孔插头接口和继承两个孔插头类）可以提供这种转换，它提供了Request方法的实现（其内部调用的是两个孔插头，因为适配器只是一个外壳罢了，包装着两个孔插头（因为只有这样，电器才能使用），并向外界提供三个孔插头的外观，）以供客户端使用。

2.3 类图

上面实现中，因为适配器（PowerAdapter类）与源角色（TwoHole类）是继承关系，所以该适配器模式是类的适配器模式，具体对应的类图为：



2.4 对象的适配器模式

上面都是类的适配器模式的介绍，然而适配器模式还有另外一种形式——对象的适配器模式，这里就具体讲解下它的实现，实现的分析思路：既然现在适配器类不能继承TwoHole抽象类了（因为用继承就属于类的适配器了），但是适配器类无论如何都要实现客户端期待的方法的，即Request方法，所以一定是要继承ThreeHole抽象类或IThreeHole接口的，然而适配器类的Request方法又必须调用TwoHole的SpecificRequest方法，又不能用继承，这时候就想，不能继承，但是我们可以在适配器类中创建TwoHole对象，然后在Requst中使用TwoHole的方法了。正如我们分析的那样，对象的适配器模式的实现正式如此。下面就让我看看具体实现代码：

```
namespace 对象的适配器模式
{
    class Client
    {
        static void Main(string[] args)
        {
            // 现在客户端可以通过电适配要使用2个孔的插头了
            ThreeHole threehole = new PowerAdapter();
            threehole.Request();
            Console.ReadLine();
        }
    }

    /// <summary>
    /// 三个孔的插头，也就是适配器模式中的目标(Target)角色
    /// </summary>
    public class ThreeHole
    {
        // 客户端需要的方法
        public virtual void Request()
        {
```

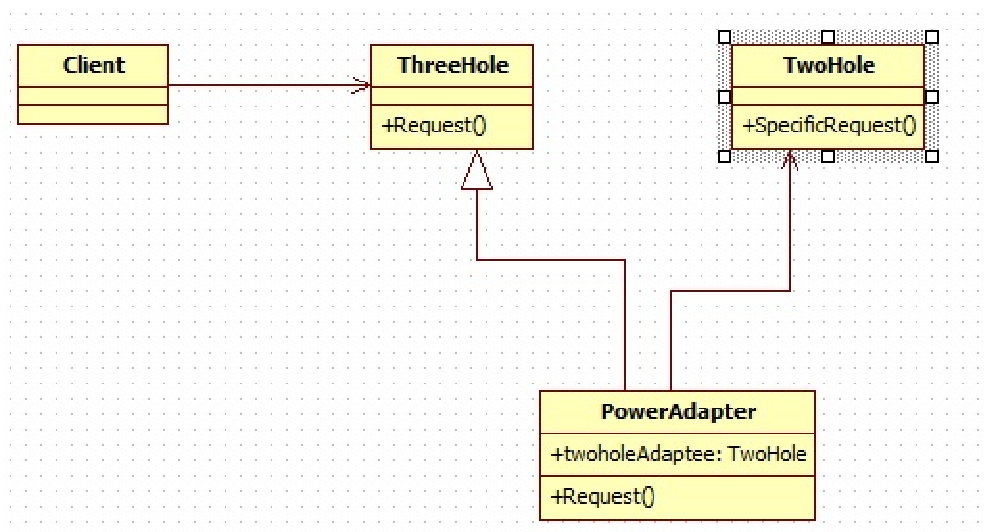
```
                // 可以把一般实现放在这里
            }
        }

        /// <summary>
        /// 两个孔的插头，源角色——需要适配的类
        /// </summary>
        public class TwoHole
        {
            public void SpecificRequest()
            {
                Console.WriteLine("我是两个孔的插头");
            }
        }

        /// <summary>
        /// 适配器类，这里适配器类没有TwoHole类，
        /// 而是引用了TwoHole对象，所以是对象的适配器模式的实现
        /// </summary>
        public class PowerAdapter : ThreeHole
        {
            // 引用两个孔插头的实例,从而将客户端与TwoHole联系起来
            public TwoHole twoholeAdaptee = new TwoHole();

            /// <summary>
            /// 实现三个孔插头接口方法
            /// </summary>
            public override void Request()
            {
                twoholeAdaptee.SpecificRequest();
            }
        }
    }
```

从上面代码可以看出,对象的适配器模式正如我们开始分析的思路去实现的, 其中客户端调用代码和类的适配器实现基本相同,下面让我们看看对象的适配器模式的类图,具体类图如下:



三、适配器模式的优缺点

在引言部分已经提出,适配器模式用来解决现有对象与客户端期待接口不一致的问题,下面详细总结下适配器两种形式的优缺点。

类的适配器模式：

优点：

- 可以在不修改原有代码的基础上来复用现有类，很好地符合 “开闭原则”
- 可以重新定义Adaptee(被适配的类)的部分行为，因为在类适配器模式中，Adapter是Adaptee的子类
- 仅仅引入一个对象，并不需要额外的字段来引用Adaptee实例（这个即是优点也是缺点）。

缺点：

- 用一个具体的Adapter类对Adaptee和Target进行匹配，当如果想要匹配一个类以及所有它的子类时，类的适配器模式就不能胜任了。因为类的适配器模式中没有引入Adaptee的实例，光调用this.SpecificRequest方法并不能去调用它对应子类的SpecificRequest方法。
- 采用了 “多继承”的实现方式，带来了不良的高耦合。

对象的适配器模式

优点：

- 可以在不修改原有代码的基础上来复用现有类，很好地符合 “开闭原则”（这点是两种实现方式都具有的）
- 采用 “对象组合”的方式，更符合松耦合。

缺点：

- 使得重定义Adaptee的行为较困难，这就需要生成Adaptee的子类并且使得Adapter引用这个子类而不是引用Adaptee本身。

四、使用场景

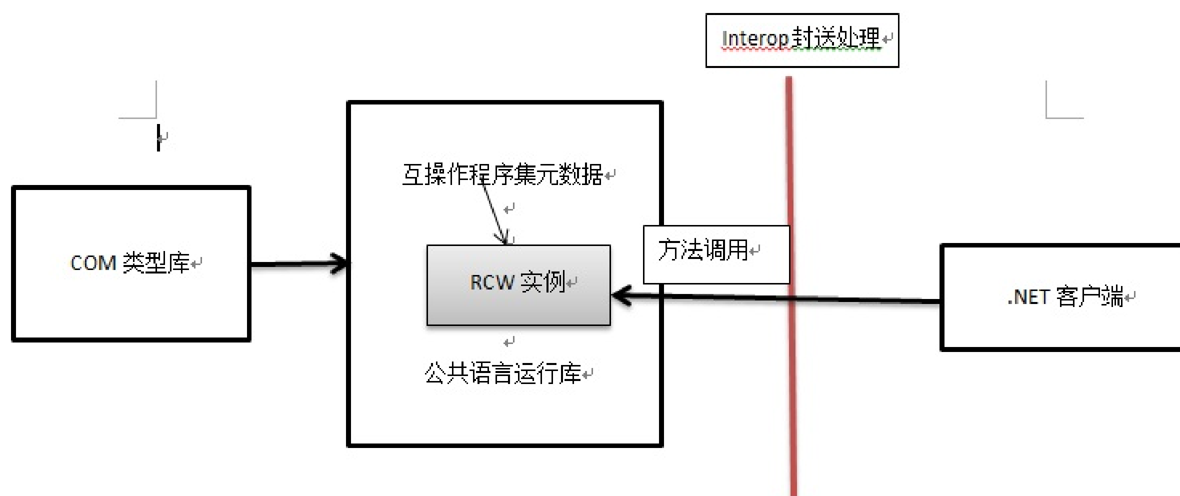
在以下情况下可以考虑使用适配器模式：

1. 系统需要复用现有类，而该类的接口不符合系统的需求
2. 想要建立一个可重复使用的类，用于与一些彼此之间没有太大关联的一些类，包括一些可能在将来引进的类一起工作。
3. 对于对象适配器模式，在设计里需要改变多个已有子类的接口，如果使用类的适配器模式，就要针对每一个子类做一个适配器，而这不太实际。

五、.NET中适配器模式的实现

1．适配器模式在.NET Framework中的一个最大的应用就是**COM Interop**。COM Interop就好像是COM和.NET之间的一座桥梁（关于COM互操作更多内容可以参考我的互操作系列）。COM组件对象与.NET类对象是完全不同的，但为了使.NET程序

象使用.NET对象一样使用COM组件，微软在处理方式上采用了Adapter模式，对COM对象进行包装，这个包装类就是RCW(Runtime Callable Wrapper)。RCW实际上是runtime生成的一个.NET类，它包装了COM组件的方法，并内部实现对COM组件的调用。如下图所示：



2．.NET中的另外一个适配器模式的应用就是**DataAdapter**。ADO.NET为统一的数据访问提供了多个接口和基类，其中最重要的接口之一是IdataAdapter。DataAdpter起到了数据库到DataSet桥接器的作用，使应用程序的数据操作统一到DataSet上，而与具体的数据库类型无关。甚至可以针对特殊的数据源编制自己的DataAdpter，从而使我们的应用程序与这些特殊的数据源相兼容。

六、总结

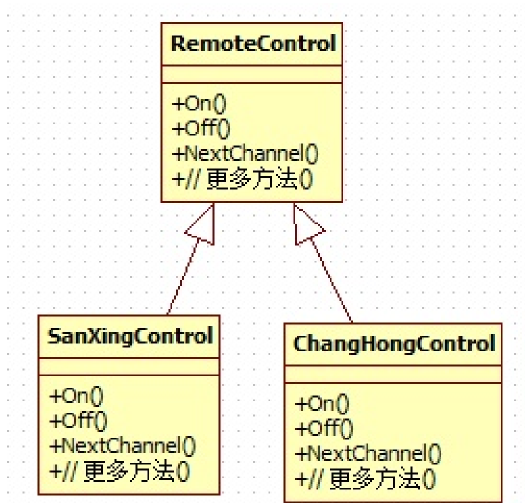
到这里适配器模式的介绍就结束了，本文主要介绍了适配器模式的两种实现、分析它们的优缺点以及使用场景的介绍，在适配器模式中，适配器可以是抽象类，并适配器模式的实现是非常灵活的，我们完全可以将**Adapter**模式中的“现存对象”作为新的接口方法参数，适配器类可以根据参数参数可以返回一个合适的实例给客户端。**

本专题的所有源码：设计模式之适配器模式

C#设计模式(8)——桥接模式（Bridge Pattern）

一、引言

这里以电视遥控器的一个例子来引出桥接模式解决的问题，首先，我们每个牌子的电视机都有一个遥控器，此时我们能想到的一个设计是——把遥控器做为一个抽象类，抽象类中提供遥控器的所有实现，其他具体电视品牌的遥控器都继承这个抽象类，具体设计类图如下：



这样的实现使得每部不同型号的电视都有自己遥控器实现，这样的设计对于电视机的改变可以很好地应对，只需要添加一个派生类就搞定了，但随着时间的推移，用户需要改变遥控器的功能，如：用户可能后面需要对遥控器添加返回上一个台等功能时，此时上面的设计就需要修改抽象类RemoteControl的提供的接口了，此时可能只需要向抽象类中添加一个方法就可以解决了，但是这样带来的问题是我们改变了抽象的实现，如果用户需要同时改变电视机品型号和遥控器功能时，上面的设计就会导致相当大的修改，显然这样的设计并不是好的设计。然而使用桥接模式可以很好地解决这个问题，下面让我具体看看桥接模式是如何实现的。

二、桥接模式的详细介绍

2.1 定义

桥接模式即将抽象部分与实现部分脱耦，使它们可以独立变化。对于上面的问题中，抽象化也就是RemoteControl类，实现部分也就是On()、Off()、NextChannel()等这样的方法（即遥控器的实现），上面的设计中，抽象化和实现部分在一起，桥接模式的目的就是使两者分离，根据面向对象的封装变化的原则，我们可以把实现部分的变化（也就是遥控器功能的变化）封装到另外一个类中，这样的一个思路也就是桥接模式的实现，大家可以对照桥接模式的实现代码来解决我们的分析思路。

2.2 桥接模式实现

上面定义部分已经给出了我们桥接模式的目的以及实现思路了，下面让我们具体看看桥接模式是如何解决引言部分设计的不足。

抽象化部分的代码：

```
/// <summary>
    /// 抽象概念中的遥控器，扮演抽象化角色
    /// </summary>
    public class RemoteControl
    {
        // 字段
        private TV implementor;

        // 属性
        public TV Implementor
        {
            get { return implementor; }
            set { implementor = value; }
        }

        /// <summary>
        /// 开电视机，这里抽象类中不再提供实现了，而是调用实现类中的实现
        /// </summary>
        public virtual void On()
        {
            implementor.On();
        }

        /// <summary>
        /// 关电视机
        /// </summary>
        public virtual void Off()
        {
            implementor.Off();
        }

        /// <summary>
        /// 换频道
        /// </summary>
        public virtual void SetChannel()
        {
            implementor.tuneChannel();
        }
    }

    /// <summary>
    /// 具体遥控器
    /// </summary>
    public class ConcreteRemote : RemoteControl
    {
```

```
        public override void SetChannel()
        {
            Console.WriteLine("---------------------");
            base.SetChannel();
            Console.WriteLine("---------------------");
        }
    }
```

遥控器的实现方法部分代码，即实现化部分代码，此时我们用另外一个抽象类TV封装了遥控器功能的变化，具体实现交给具体型号电视机去完成：

```
/// <summary>
    /// 电视机，提供抽象方法
    /// </summary>
    public abstract class TV
    {
        public abstract void On();
        public abstract void Off();
        public abstract void tuneChannel();
    }

    /// <summary>
    /// 长虹牌电视机，重写基类的抽象方法
    /// 提供具体的实现
    /// </summary>
    public class ChangHong : TV
    {
        public override void On()
        {
            Console.WriteLine("长虹牌电视机已经打开了");
        }

        public override void Off()
        {
            Console.WriteLine("长虹牌电视机已经关掉了");
        }

        public override void tuneChannel()
        {
            Console.WriteLine("长虹牌电视机换频道");
        }
    }

    /// <summary>
    /// 三星牌电视机，重写基类的抽象方法
    /// </summary>
    public class Samsung : TV
    {
        public override void On()
        {
            Console.WriteLine("三星牌电视机已经打开了");
```

```
        }

        public override void Off()
        {
            Console.WriteLine("三星牌电视机已经关掉了");
        }

        public override void tuneChannel()
        {
            Console.WriteLine("三星牌电视机换频道");
        }
    }
```

采用桥接模式的客户端调用代码：

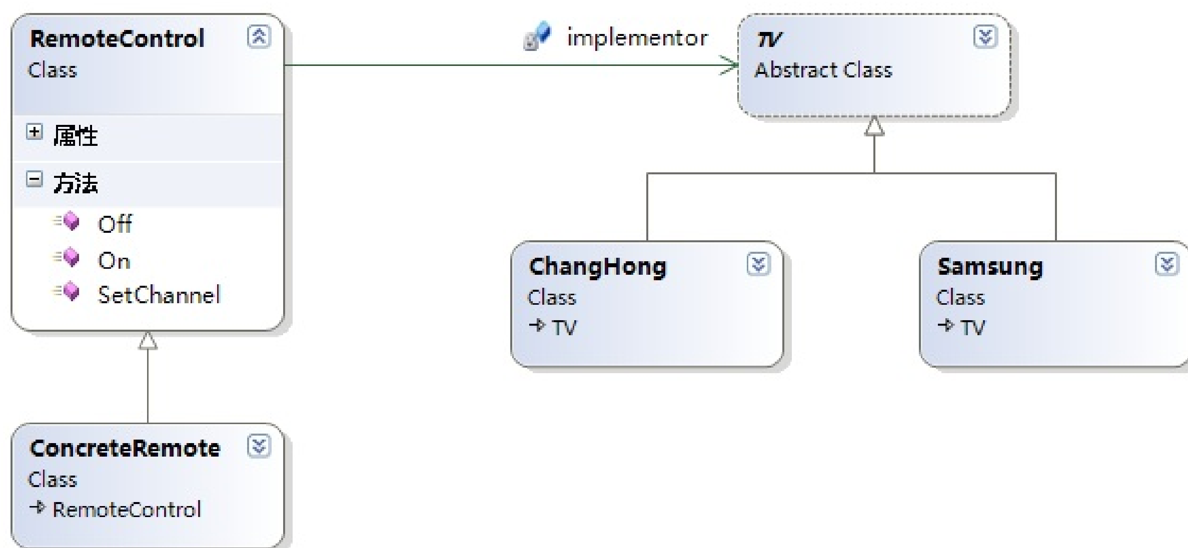
```
/// <summary>
    /// 以电视机遥控器的例子来演示桥接模式
    /// </summary>
    class Client
    {
        static void Main(string[] args)
        {
            // 创建一个遥控器
            RemoteControl remoteControl = new ConcreteRemote();
            // 长虹电视机
            remoteControl.Implementor = new ChangHong();
            remoteControl.On();
            remoteControl.SetChannel();
            remoteControl.Off();
            Console.WriteLine();

            // 三星牌电视机
            remoteControl.Implementor = new Samsung();
            remoteControl.On();
            remoteControl.SetChannel();
            remoteControl.Off();
            Console.Read();
        }
    }
```

上面桥接模式的实现中，遥控器的功能实现方法不在遥控器抽象类中去实现了，而是把实现部分用来另一个电视机类去封装它，然而遥控器中只包含电视机类的一个引用，同时这样的设计也非常符合现实生活中的情况（我认为的现实生活中遥控器的实现——遥控器中并不包含换台，打开电视机这样的功能的实现，遥控器只是包含了电视机上这些功能的引用，然后红外线去找到电视机上对应功能的的实现）。通过桥接模式，我们把抽象化和实现化部分分离开了，这样就可以很好应对这两方面的变化了。

2.3 桥接模式的类图

看完桥接模式的实现后，为了帮助大家理清对桥接模式中类之间关系，这里给出桥接模式的类图结构：



三、桥接模式的优缺点

介绍完桥接模式，让我们看看桥接模式具体哪些优缺点。

优点：

把抽象接口与其实现解耦。

抽象和实现可以独立扩展，不会影响到对方。

实现细节对客户透明，对用于隐藏了具体实现细节。

缺点：增加了系统的复杂度

四、使用场景

我们再来看看桥接模式的使用场景，在以下情况下应当使用桥接模式：

1. 如果一个系统需要在构件的抽象化角色和具体化角色之间添加更多的灵活性，避免在两个层次之间建立静态的联系。
2. 设计要求实现化角色的任何改变不应当影响客户端，或者实现化角色的改变对客户端是完全透明的。
3. 需要跨越多个平台的图形和窗口系统上。
4. 一个类存在两个独立变化的维度，且两个维度都需要进行扩展。

五、一个实际应用桥接模式的例子

桥接模式也经常用于具体的系统开发中，对于三层架构中就应用了桥接模式，三层架构中的业务逻辑层BLL中通过桥接模式与数据操作层解耦（DAL），其实现方式就是在BLL层中引用了DAL层中一个引用。这样数据操作的实现可以在不改变客户端代码的情况下动态进行更换，下面看一个简单的示例代码：

```
// 客户端调用
    // 类似Web应用程序
    class Client
    {
        static void Main(string[] args)
        {
            BusinessObject customers = new CustomersBusinessObject(
            customers.Dataacces = new CustomersDataAccess();

            customers.Add("小六");
            Console.WriteLine("增加了一位成员的结果：");
            customers.ShowAll();
            customers.Delete("王五");
            Console.WriteLine("删除了一位成员的结果：");
            customers.ShowAll();
            Console.WriteLine("更新了一位成员的结果：");
            customers.Update("Learning_Hard");
            customers.ShowAll();

            Console.Read();
        }
    }

    // BLL 层
    public class BusinessObject
    {
        // 字段
        private DataAccess dataacess;
        private string city;

        public BusinessObject(string city)
        {
            this.city = city;
        }

        // 属性
        public DataAccess Dataacces
        {
            get { return dataacess; }
            set { dataacess = value; }
        }

        // 方法
        public virtual void Add(string name)
        {
            Dataacces.AddRecord(name);
        }
```

```
        public virtual void Delete(string name)
        {
            Dataacces.DeleteRecord(name);
        }

        public virtual void Update(string name)
        {
            Dataacces.UpdateRecord(name);
        }

        public virtual string Get(int index)
        {
            return Dataacces.GetRecord(index);
        }
        public virtual void ShowAll()
        {
            Console.WriteLine();
            Console.WriteLine("{0}的顾客有：", city);
            Dataacces.ShowAllRecords();
        }
    }

    public class CustomersBusinessObject : BusinessObject
    {
        public CustomersBusinessObject(string city)
            : base(city) { }

        // 重写方法
        public override void ShowAll()
        {
            Console.WriteLine("------------------------");
            base.ShowAll();
            Console.WriteLine("------------------------");
        }
    }

    /// <summary>
    /// 相当于三层架构中数据访问层（DAL）
    /// </summary>
    public abstract class DataAccess
    {
        // 对记录的增删改查操作
        public abstract void AddRecord(string name);
        public abstract void DeleteRecord(string name);
        public abstract void UpdateRecord(string name);
        public abstract string GetRecord(int index);
        public abstract void ShowAllRecords();
    }

    public class CustomersDataAccess:DataAccess
    {
        // 字段
```

```
        private List<string> customers =new List<string>();

        public CustomersDataAccess()
        {
            // 实际业务中从数据库中读取数据再填充列表
            customers.Add("Learning Hard");
            customers.Add("张三");
            customers.Add("李四");
            customers.Add("王五");
        }
        // 重写方法
        public override void AddRecord(string name)
        {
            customers.Add(name);
        }

        public override void DeleteRecord(string name)
        {
            customers.Remove(name);
        }

        public override void UpdateRecord(string updatename)
        {
            customers[0] = updatename;
        }

        public override string GetRecord(int index)
        {
            return customers[index];
        }

        public override void ShowAllRecords()
        {
            foreach (string name in customers)
            {
                Console.WriteLine(" " + name);
            }
        }

    }
```

六、总结

到这里，桥接模式的介绍就介绍，桥接模式实现了抽象化与实现化的解耦，使它们相互独立互不影响到对方。

C#设计模式(9)——装饰者模式（Decorator Pattern）

一、引言

在软件开发中，我们经常想要对一类对象添加不同的功能，例如要给手机添加贴膜，手机挂件，手机外壳等，如果此时利用继承来实现的话，就需要定义无数的类，如StickerPhone（贴膜是手机类）、AccessoriesPhone（挂件手机类）等，这样就会导致 ”子类爆炸“问题，为了解决这个问题，我们可以使用装饰者模式来动态地给一个对象添加额外的职责。下面让我们看看装饰者模式。

二、装饰者模式的详细介绍

2.1 定义

装饰者模式以对客户透明的方式动态地给一个对象附加上更多的责任，装饰者模式相比生成子类可以更灵活地增加功能。

2.2 装饰者模式实现

这里以手机和手机配件的例子来演示装饰者模式的实现，具体代码如下：

```
/// <summary>
    /// 手机抽象类，即装饰者模式中的抽象组件类
    /// </summary>
    public abstract class Phone
    {
        public abstract void Print();
    }

    /// <summary>
    /// 苹果手机，即装饰着模式中的具体组件类
    /// </summary>
    public class ApplePhone:Phone
    {
        /// <summary>
        /// 重写基类方法
        /// </summary>
        public override void Print()
        {
            Console.WriteLine("开始执行具体的对象——苹果手机");
        }
    }
```

```
    /// <summary>
    /// 装饰抽象类,要让装饰完全取代抽象组件，所以必须继承自Photo
    /// </summary>
    public abstract class Decorator:Phone
    {
        private Phone phone;

        public Decorator(Phone p)
        {
            this.phone = p;
        }

        public override void Print()
        {
            if (phone != null)
            {
                phone.Print();
            }
        }
    }

    /// <summary>
    /// 贴膜，即具体装饰者
    /// </summary>
    public class Sticker : Decorator
    {
        public Sticker(Phone p)
            : base(p)
        {
        }

        public override void Print()
        {
            base.Print();

            // 添加新的行为
            AddSticker();
        }

        /// <summary>
        /// 新的行为方法
        /// </summary>
        public void AddSticker()
        {
            Console.WriteLine("现在苹果手机有贴膜了");
        }
    }

    /// <summary>
    /// 手机挂件
    /// </summary>
    public class Accessories : Decorator
    {
```

```
        public Accessories(Phone p)
            : base(p)
        {
        }

        public override void Print()
        {
            base.Print();

            // 添加新的行为
            AddAccessories();
        }

        /// <summary>
        /// 新的行为方法
        /// </summary>
        public void AddAccessories()
        {
            Console.WriteLine("现在苹果手机有漂亮的挂件了");
        }
    }
```

此时客户端调用代码如下：

```
class Customer
    {
        static void Main(string[] args)
        {
            // 我买了个苹果手机
            Phone phone = new ApplePhone();

            // 现在想贴膜了
            Decorator applePhoneWithSticker = new Sticker(phone);
            // 扩展贴膜行为
            applePhoneWithSticker.Print();
            Console.WriteLine("----------------------\n");

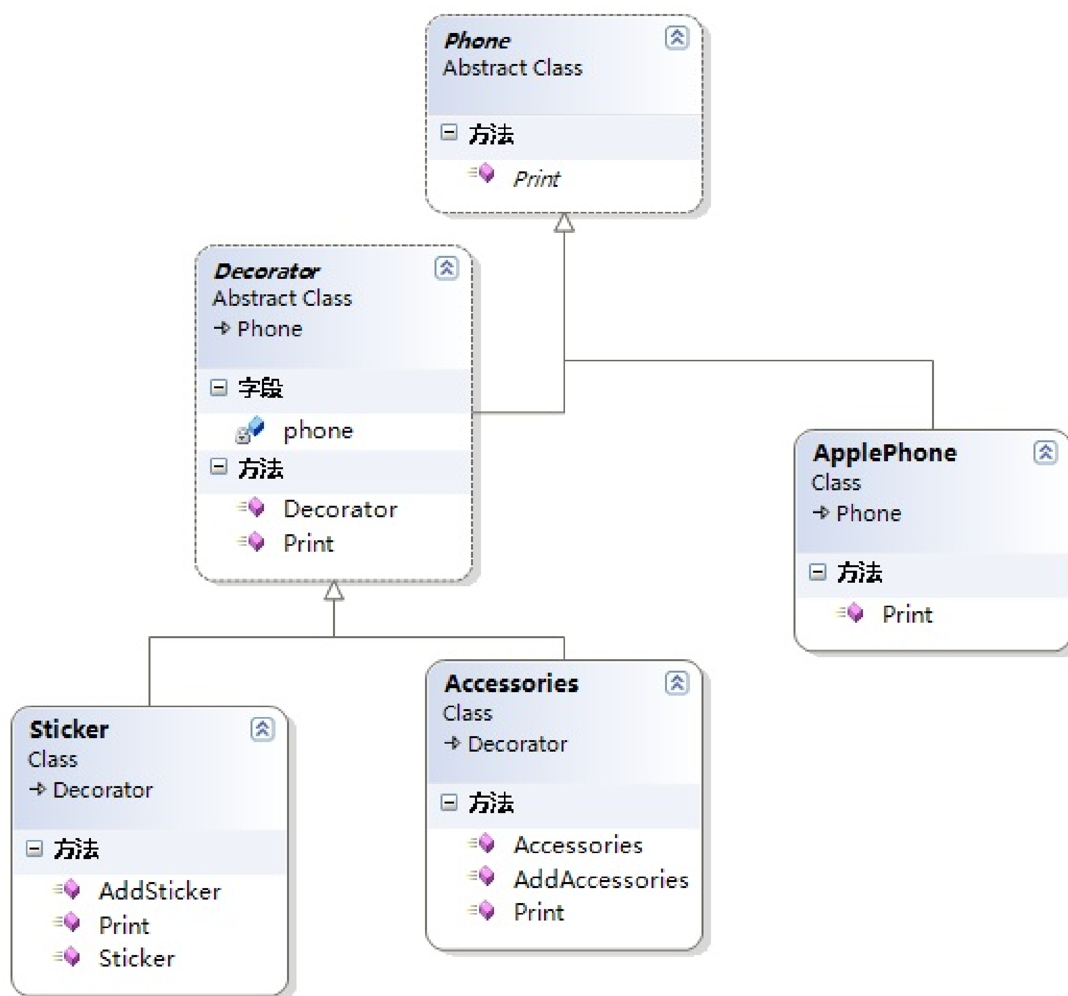
            // 现在我想有挂件了
            Decorator applePhoneWithAccessories = new Accessories(p
            // 扩展手机挂件行为
            applePhoneWithAccessories.Print();
            Console.WriteLine("----------------------\n");

            // 现在我同时有贴膜和手机挂件了
            Sticker sticker = new Sticker(phone);
            Accessories applePhoneWithAccessoriesAndSticker = new A
            applePhoneWithAccessoriesAndSticker.Print();
            Console.ReadLine();
        }
```

从上面的客户端代码可以看出，客户端可以动态地将手机配件增加到手机上，如果需要添加手机外壳时，此时只需要添加一个继承Decorator的手机外壳类，从而，装饰者模式扩展性也非常好。

2.3 装饰者模式的类图

实现完了装饰者模式之后，让我们看看装饰者模式实现中类之间的关系，具体见下图：



在装饰者模式中各个角色有：

- 抽象构件（Phone）角色：给出一个抽象接口，以规范准备接受附加责任的对象。
- 具体构件（AppPhone）角色：定义一个将要接收附加责任的类。
- 装饰（Dicorator）角色：持有一个构件（Component）对象的实例，并定义一个与抽象构件接口一致的接口。
- 具体装饰（Sticker和Accessories）角色：负责给构件对象"贴上"附加的责任。

三、装饰者模式的优缺点

看完装饰者模式的详细介绍之后，我们继续分析下它的优缺点。

优点：

1. 装饰这模式和继承的目的都是扩展对象的功能，但装饰者模式比继承更灵活
2. 通过使用不同的具体装饰类以及这些类的排列组合，设计师可以创造出很多不同行为的组合
3. 装饰者模式有很好地可扩展性

缺点：装饰者模式会导致设计中出现许多小对象，如果过度使用，会让程序变的更复杂。并且更多的对象会是的差错变得困难，特别是这些对象看上去都很像。

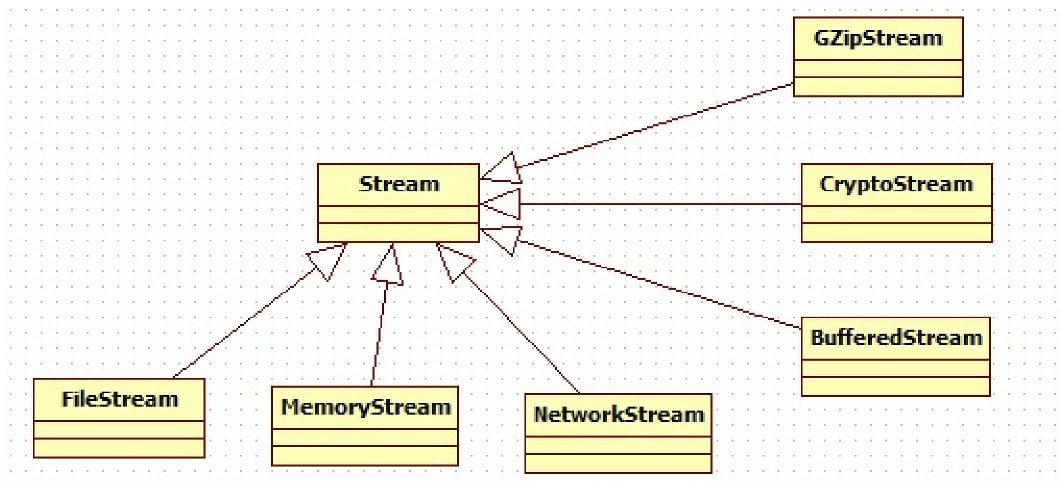
四、使用场景

下面让我们看看装饰者模式具体在哪些情况下使用，在以下情况下应当使用装饰者模式：

1. 需要扩展一个类的功能或给一个类增加附加责任。
2. 需要动态地给一个对象增加功能，这些功能可以再动态地撤销。
3. 需要增加由一些基本功能的排列组合而产生的非常大量的功能

五、.NET中装饰者模式的实现

在.NET 类库中也有装饰者模式的实现，该类就是System.IO.Stream,下面看看Stream类结构：



上图中，BufferedStream、CryptoStream和GZipStream其实就是两个具体装饰类，这里的装饰者模式省略了抽象装饰角色（Decorator）。下面演示下客户端如何动态地为MemoryStream动态增加功能的。

```
MemoryStream memoryStream = new MemoryStream(new byte[] {95,96,97,

            // 扩展缓冲的功能
            BufferedStream buffStream = new BufferedStream(memorySt

            // 添加加密的功能
            CryptoStream cryptoStream = new CryptoStream(memoryStre
            // 添加压缩功能
            GZipStream gzipStream = new GZipStream(memoryStream, Co
```

六、总结

到这里，装饰者模式的介绍就结束了，装饰者模式采用对象组合而非继承的方式实现了再运行时动态地扩展对象功能的能力，而且可以根据需要扩展多个功能，避免了单独使用继承带来的 ”灵活性差“和”多子类衍生问题“。同时它很好地符合面向对象设计原则中 ”优先使用对象组合而非继承“和”开放-封闭“原则。

本专题所有源码：设计模式之装饰者模式

C#设计模式(10)——组合模式（Composite Pattern）

一、引言

在软件开发过程中，我们经常会遇到处理简单对象和复合对象的情况，例如对操作系统中目录的处理就是这样的一个例子，因为目录可以包括单独的文件，也可以包括文件夹，文件夹又是由文件组成的，由于简单对象和复合对象在功能上区别，导致在操作过程中必须区分简单对象和复合对象，这样就会导致客户调用带来不必要的麻烦，然而作为客户，它们希望能够始终一致地对待简单对象和复合对象。然而组合模式就是解决这样的问题。下面让我们看看组合模式是怎样解决这个问题的。

二、组合模式的详细介绍

2.1 组合模式的定义

组合模式允许你将对象组合成树形结构来表现”部分-整体“的层次结构，使得客户以一致的方式处理单个对象以及对象的组合。下面我们用绘制的例子来详细介绍组合模式，图形可以由一些基本图形元素组成（如直线，圆等），也可以由一些复杂图形组成（由基本图形元素组合而成），为了使客户对基本图形和复杂图形的调用保持一致，我们使用组合模式来达到整个目的。

组合模式实现的最关键的地方是——简单对象和复合对象必须实现相同的接口。这就是组合模式能够将组合对象和简单对象进行一致处理的原因。

2.2 组合模式的实现

介绍完组合模式的定义之后，让我们以图形的例子来实现组合模式，具体代码如下：

```
// 通过一些简单图形以及一些复杂图形构建图形树来演示组合模式
    // 客户端调用
    class Client
    {
        static void Main(string[] args)
        {
            ComplexGraphics complexGraphics = new ComplexGraphics('
            complexGraphics.Add(new Line("线段A"));
            ComplexGraphics CompositeCG = new ComplexGraphics("一个
            CompositeCG.Add(new Circle("圆"));
            CompositeCG.Add(new Circle("线段B"));
            complexGraphics.Add(CompositeCG);
            Line l = new Line("线段C");
```

```
            complexGraphics.Add(l);

            // 显示复杂图形的画法
            Console.WriteLine("复杂图形的绘制如下：");
            Console.WriteLine("---------------------");
            complexGraphics.Draw();
            Console.WriteLine("复杂图形绘制完成");
            Console.WriteLine("---------------------");
            Console.WriteLine();

            // 移除一个组件再显示复杂图形的画法
            complexGraphics.Remove(l);
            Console.WriteLine("移除线段C后，复杂图形的绘制如下：");
            Console.WriteLine("---------------------");
            complexGraphics.Draw();
            Console.WriteLine("复杂图形绘制完成");
            Console.WriteLine("---------------------");
            Console.Read();
        }
    }

    /// <summary>
    /// 图形抽象类，
    /// </summary>
    public abstract class Graphics
    {
        public string Name { get; set; }
        public Graphics(string name)
        {
            this.Name = name;
        }

        public abstract void Draw();
        public abstract void Add(Graphics g);
        public abstract void Remove(Graphics g);
    }

    /// <summary>
    /// 简单图形类——线
    /// </summary>
    public class Line : Graphics
    {
        public Line(string name)
            : base(name)
        { }

        // 重写父类抽象方法
        public override void Draw()
        {
            Console.WriteLine("画  " + Name);
        }
        // 因为简单图形在添加或移除其他图形，所以简单图形Add或Remove方法没有
        // 如果客户端调用了简单图形的Add或Remove方法将会在运行时抛出异常
```

```
        // 我们可以在客户端捕获该类移除并处理
        public override void Add(Graphics g)
        {
            throw new Exception("不能向简单图形Line添加其他图形");
        }
        public override void Remove(Graphics g)
        {
            throw new Exception("不能向简单图形Line移除其他图形");
        }
    }

    /// <summary>
    /// 简单图形类——圆
    /// </summary>
    public class Circle : Graphics
    {
        public Circle(string name)
            : base(name)
        { }

        // 重写父类抽象方法
        public override void Draw()
        {
            Console.WriteLine("画  " + Name);
        }

        public override void Add(Graphics g)
        {
            throw new Exception("不能向简单图形Circle添加其他图形");
        }
        public override void Remove(Graphics g)
        {
            throw new Exception("不能向简单图形Circle移除其他图形");
        }
    }

    /// <summary>
    /// 复杂图形，由一些简单图形组成,这里假设该复杂图形由一个圆两条线组成的复杂
    /// </summary>
    public class ComplexGraphics : Graphics
    {
        private List<Graphics> complexGraphicsList = new List<Graph

        public ComplexGraphics(string name)
            : base(name)
        { }

        /// <summary>
        /// 复杂图形的画法
        /// </summary>
        public override void Draw()
        {
            foreach (Graphics g in complexGraphicsList)
```

```
            {
                g.Draw();
            }
        }

        public override void Add(Graphics g)
        {
            complexGraphicsList.Add(g);
        }
        public override void Remove(Graphics g)
        {
            complexGraphicsList.Remove(g);
        }
    }
```

由于基本图形对象不存在Add和Remove方法，上面实现中直接通过抛出一个异常的方式来解决这样的问题的，但是我们想以一种更安全的方式来解决——因为基本图形根本不存在这样的方法，我们是不是可以移除这些方法呢？为了移除这些方法，我们就不得不修改Graphics接口，我们把管理子对象的方法声明放在复合图形对象里面，这样简单对象Line、Circle使用这些方法时在编译时就会出错，这样的一种实现方式我们称为安全式的组合模式，然而上面的实现方式称为透明式的组合模式，下面让我们看看安全式的组合模式又是怎样实现的，具体实现代码如下：

```
/// 安全式的组合模式
    /// 此方式实现的组合模式把管理子对象的方法声明在树枝构件ComplexGraphics
    /// 这样如果叶子节点Line、Circle使用了Add或Remove方法时，就能在编译期间
    /// 但这种方式虽然解决了透明式组合模式的问题，但是它使得叶子节点和树枝构件
    /// 所以这两种方式实现的组合模式各有优缺点，具体使用哪个，可以根据问题的实
    class Client
    {
        static void Main(string[] args)
        {
            ComplexGraphics complexGraphics = new ComplexGraphics('
            complexGraphics.Add(new Line("线段A"));
            ComplexGraphics CompositeCG = new ComplexGraphics("一个
            CompositeCG.Add(new Circle("圆"));
            CompositeCG.Add(new Circle("线段B"));
            complexGraphics.Add(CompositeCG);
            Line l = new Line("线段C");
            complexGraphics.Add(l);

            // 显示复杂图形的画法
            Console.WriteLine("复杂图形的绘制如下：");
            Console.WriteLine("---------------------");
            complexGraphics.Draw();
            Console.WriteLine("复杂图形绘制完成");
            Console.WriteLine("---------------------");
            Console.WriteLine();

            // 移除一个组件再显示复杂图形的画法
```

```
            complexGraphics.Remove(l);
            Console.WriteLine("移除线段C后，复杂图形的绘制如下：");
            Console.WriteLine("---------------------");
            complexGraphics.Draw();
            Console.WriteLine("复杂图形绘制完成");
            Console.WriteLine("---------------------");
            Console.Read();
        }
    }

    /// <summary>
    /// 图形抽象类，
    /// </summary>
    public abstract class Graphics
    {
        public string Name { get; set; }
        public Graphics(string name)
        {
            this.Name = name;
        }

        public abstract void Draw();
        // 移除了Add和Remove方法
        // 把管理子对象的方法放到了ComplexGraphics类中进行管理
        // 因为这些方法只在复杂图形中才有意义
    }

    /// <summary>
    /// 简单图形类——线
    /// </summary>
    public class Line : Graphics
    {
        public Line(string name)
            : base(name)
        { }

        // 重写父类抽象方法
        public override void Draw()
        {
            Console.WriteLine("画  " + Name);
        }
    }

    /// <summary>
    /// 简单图形类——圆
    /// </summary>
    public class Circle : Graphics
    {
        public Circle(string name)
            : base(name)
        { }

        // 重写父类抽象方法
```

```
        public override void Draw()
        {
            Console.WriteLine("画  " + Name);
        }
    }

    /// <summary>
    /// 复杂图形，由一些简单图形组成,这里假设该复杂图形由一个圆两条线组成的复杂
    /// </summary>
    public class ComplexGraphics : Graphics
    {
        private List<Graphics> complexGraphicsList = new List<Graph
        public ComplexGraphics(string name)
            : base(name)
        { }

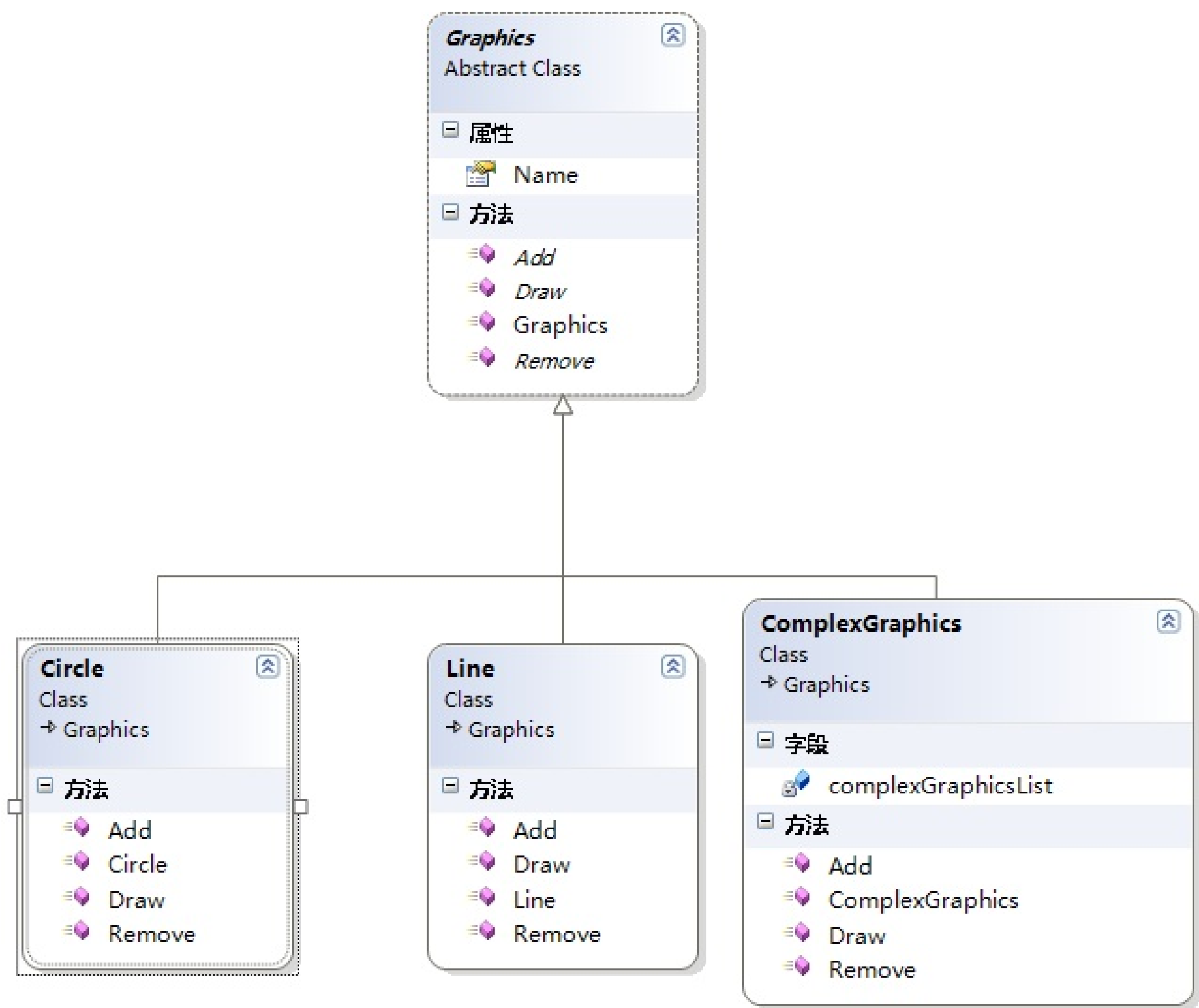
        /// <summary>
        /// 复杂图形的画法
        /// </summary>
        public override void Draw()
        {
            foreach (Graphics g in complexGraphicsList)
            {
                g.Draw();
            }
        }

        public void Add(Graphics g)
        {
            complexGraphicsList.Add(g);
        }
        public void Remove(Graphics g)
        {
            complexGraphicsList.Remove(g);
        }
    }
```

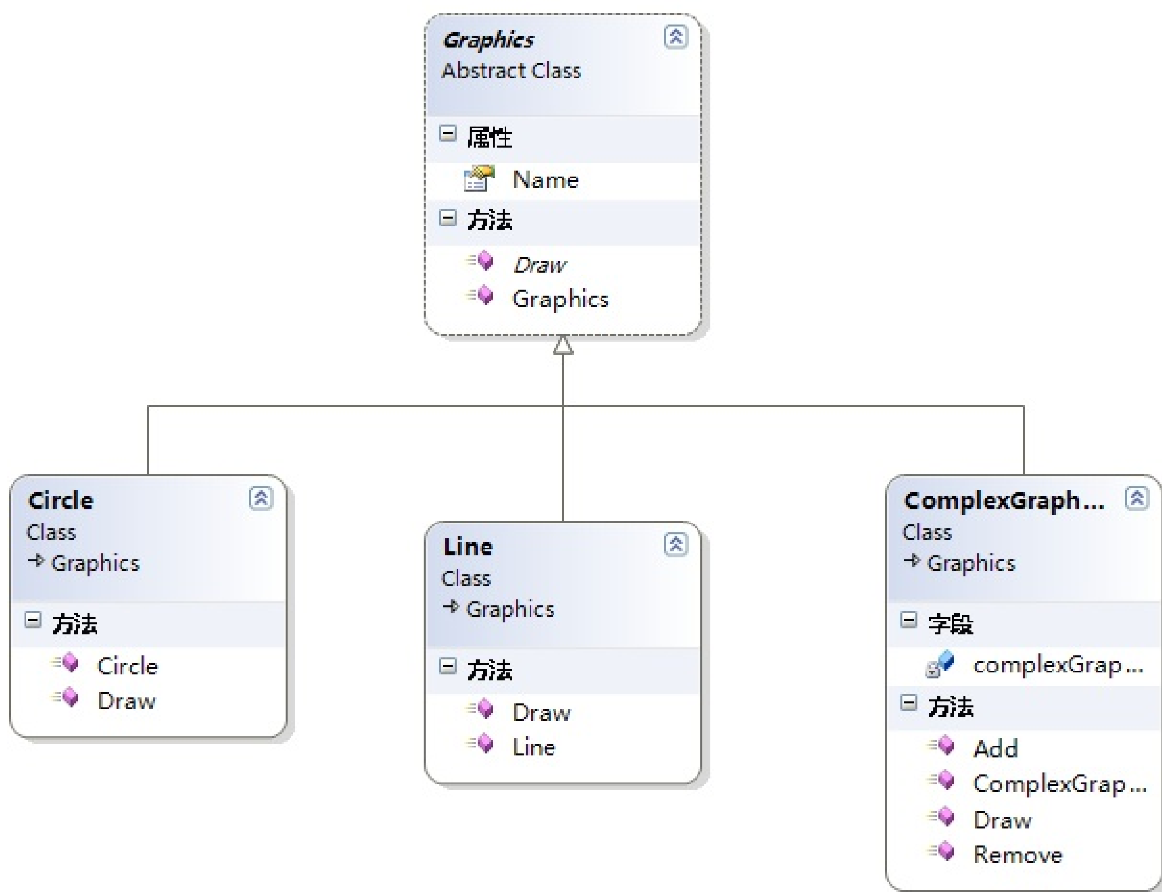
2.3 组合模式的类图

看完了上面两者方式的实现之后，让我们具体看看组合模式的类图来理清楚组合模式中类之间的关系。

透明式的组合模式类图：


Graphics
Abstract Class
属性
Name
方法
Add
Draw
Graphics
Remove
Circle
Class
→ Graphics
方法
Add
Circle
Draw
Remove
Line
Class
→ Graphics
方法
Add
Draw
Line
Remove
ComplexGraphics
Class
→ Graphics
字段
complexGraphicsList
方法
Add
ComplexGraphics
Draw
Remove

安全式组合模式的类图：



组合模式中涉及到三个角色：

- 抽象构件（**Component**）角色：这是一个抽象角色，上面实现中**Graphics**充当这个角色，它给参加组合的对象定义出了公共的接口及默认行为，可以用来管理所有的子对象（在透明式的组合模式是这样的）。在安全式的组合模式里，构件角色并不定义出管理子对象的方法，这一定义由树枝结构对象给出。
- 树叶构件（**Leaf**）角色：树叶对象时没有下级子对象的对象，上面实现中**Line**和**Circle**充当这个角色，定义出参加组合的原始对象的行为
- 树枝构件（**Composite**）角色：代表参加组合的有下级子对象的对象，上面实现中**ComplexGraphics**充当这个角色，树枝对象给出所有管理子对象的方法实现，如Add、Remove等。

三、组合模式的优缺点

优点：

1. 组合模式使得客户端代码可以一致地处理对象和对象容器，无需关系处理的单个对象，还是组合的对象容器。
2. 将”客户代码与复杂的对象容器结构“解耦。
3. 可以更容易地往组合对象中加入新的构件。

缺点：使得设计更加复杂。客户端需要花更多时间理清类之间的层次关系。（这个是几乎所有设计模式所面临的问题）。

注意的问题：

1. 有时候系统需要遍历一个树枝结构的子构件很多次，这时候可以考虑把遍历子构件的结构存储在父构件里面作为缓存。
2. 客户端尽量不要直接调用树叶类中的方法（在我上面实现就是这样的，创建的是一个树枝的具体对象，应该使用**Graphics** complexGraphics = new ComplexGraphics("一个复杂图形和两条线段组成的复杂图形");），而是借用其父类（Graphics）的多态性完成调用，这样可以增加代码的复用性。

四、组合模式的使用场景

在以下情况下应该考虑使用组合模式：

1. 需要表示一个对象整体或部分的层次结构。
2. 希望用户忽略组合对象与单个对象的不同，用户将统一地使用组合结构中的所有对象。

五、组合模式在.NET中的应用

组合模式在.NET 中最典型的应用就是应用与WinForms和Web的开发中，在.NET类库中，都为这两个平台提供了很多现有的控件，然而System.Windows.Forms.dll中System.Windows.Forms.Control类就应用了组合模式，因为控件包括Label、TextBox等这样的简单控件，同时也包括GroupBox、DataGrid这样复合的控件，每个控件都需要调用OnPaint方法来进行控件显示，为了表示这种对象之间整体与部分的层次结构，微软把Control类的实现应用了组合模式（确切地说应用了透明式的组合模式）。

六、总结

到这里组合模式的介绍就结束了，组合模式解耦了客户程序与复杂元素内部结构，从而使客户程序可以向处理简单元素一样来处理复杂元素。

本文中所有源码：设计模式之组合模式

C#设计模式(11)——外观模式（Facade Pattern）

一、引言

在软件开发过程中，客户端程序经常会与复杂系统的内部子系统进行耦合，从而导致客户端程序随着子系统的变化而变化，然而为了将复杂系统的内部子系统与客户端之间的依赖解耦，从而就有了外观模式，也称作 ”门面“模式。下面就具体介绍下外观模式。

二、外观模式的详细介绍

2.1 定义

外观模式提供了一个统一的接口，用来访问子系统中的一群接口。外观定义了一个高层接口，让子系统更容易使用。使用外观模式时，我们创建了一个统一的类，用来包装子系统中一个或多个复杂的类，客户端可以直接通过外观类来调用内部子系统中方法，从而外观模式让客户和子系统之间避免了紧耦合。

2.2 外观模式实现

介绍了外观模式的定义之后，让我们具体看看外观模式的由来以及实现，下面与学校中一个选课系统为例来解释外观模式，例如在选课系统中，有注册课程子系统和通知子系统，在不使用外观模式的情况下，客户端必须同时保存注册课程子系统和通知子系统两个引用，如果后期这两个子系统发生改变时，此时客户端的调用代码也要随之改变，这样就没有很好的可扩展性，下面看看不使用外观模式下选课系统的实现方式和客户端调用代码：

```
/// <summary>
    /// 不使用外观模式的情况
    /// 此时客户端与三个子系统都发送了耦合，使得客户端程序依赖与子系统
    /// 为了解决这样的问题，我们可以使用外观模式来为所有子系统设计一个统一的接
    /// 客户端只需要调用外观类中的方法就可以了，简化了客户端的操作
    /// 从而让客户和子系统之间避免了紧耦合
    /// </summary>
    class Client
    {
        static void Main(string[] args)
        {
            SubSystemA a = new SubSystemA();
            SubSystemB b = new SubSystemB();
            SubSystemC c = new SubSystemC();
            a.MethodA();
            b.MethodB();
            c.MethodC();
            Console.Read();
        }
    }

    // 子系统A
    public class SubSystemA
    {
        public void MethodA()
        {
            Console.WriteLine("执行子系统A中的方法A");
        }
    }

    // 子系统B
    public class SubSystemB
    {
        public void MethodB()
        {
            Console.WriteLine("执行子系统B中的方法B");
        }
    }

    // 子系统C
    public class SubSystemC
    {
        public void MethodC()
        {
            Console.WriteLine("执行子系统C中的方法C");
        }
    }
```

然而外观模式可以解决我们上面所说的问题，下面具体看看使用外观模式的实现：

```
/// <summary>
    /// 以学生选课系统为例子演示外观模式的使用
    /// 学生选课模块包括功能有：
    /// 验证选课的人数是否已满
    /// 通知用户课程选择成功与否
    /// 客户端代码
    /// </summary>
    class Student
    {
        private static RegistrationFacade facade = new Registration

        static void Main(string[] args)
        {
            if (facade.RegisterCourse("设计模式", "Learning Hard"))
            {
                Console.WriteLine("选课成功");
            }
            else
            {
                Console.WriteLine("选课失败");
            }

            Console.Read();
        }
    }

    // 外观类
    public class RegistrationFacade
    {
        private RegisterCourse registerCourse;
        private NotifyStudent notifyStu;
        public RegistrationFacade()
        {
            registerCourse = new RegisterCourse();
            notifyStu = new NotifyStudent();
        }

        public bool RegisterCourse(string courseName, string studen
        {
            if (!registerCourse.CheckAvailable(courseName))
            {
                return false;
            }

            return notifyStu.Notify(studentName);
        }
    }

    #region 子系统
    // 相当于子系统A
    public class RegisterCourse
    {
```

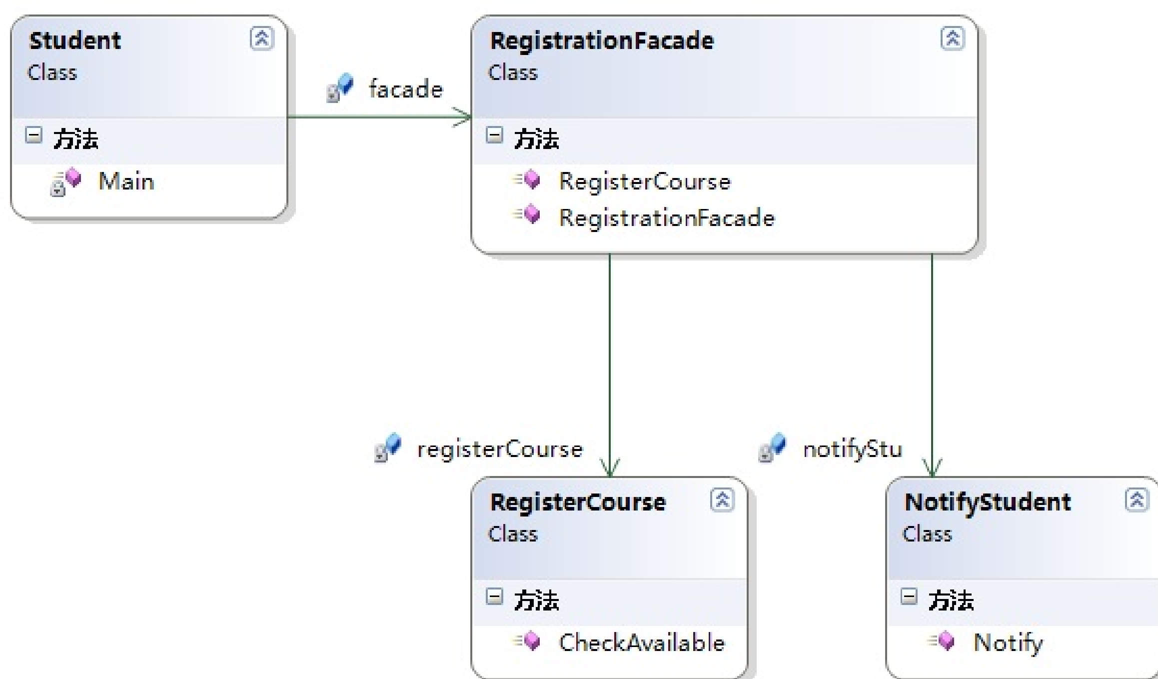
```
        public bool CheckAvailable(string courseName)
        {
            Console.WriteLine("正在验证课程 {0}是否人数已满", courseNar
            return true;
        }
    }

    // 相当于子系统B
    public class NotifyStudent
    {
        public bool Notify(string studentName)
        {
            Console.WriteLine("正在向{0}发生通知", studentName);
            return true;
        }
    }
    #endregion
```

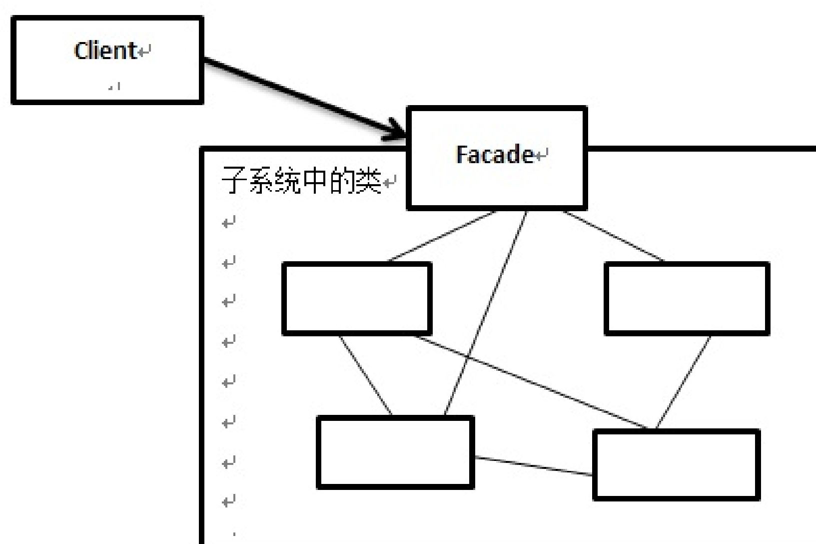
使用了外观模式之后，客户端只依赖与外观类，从而将客户端与子系统的依赖解耦了，如果子系统发生改变，此时客户端的代码并不需要去改变。外观模式的实现核心主要是——由外观类去保存各个子系统的引用，实现由一个统一的外观类去包装多个子系统类，然后客户端只需要引用这个外观类，然后由外观类来调用各个子系统中的方法。然而这样的实现方式非常类似适配器模式，然而外观模式与适配器模式不同的是：适配器模式是将一个对象包装起来以改变其接口，而外观是将一群对象 "包装"起来以简化其接口。**它们的意图是不一样的，适配器是将接口转换为不同接口，而外观模式是提供一个统一的接口来简化接口**。

2.3 外观模式的结构

看完外观模式的实现之后，为了帮助理清外观模式中类之间的关系，下面给出上面实现代码中类图：



然而对于外观模式而言，是没有一个一般化的类图描述，下面演示一个外观模式的示意性对象图来加深大家对外观模式的理解：



在上面的对象图中有两个角色：

门面（**Facade**）角色：客户端调用这个角色的方法。该角色知道相关的一个或多个子系统的功能和责任，该角色会将从客户端发来的请求委派带相应的子系统中去。

子系统（**subsystem**）角色：可以同时包含一个或多个子系统。每个子系统都不是一个单独的类，而是一个类的集合。每个子系统都可以被客户端直接调用或被门面角色调用。对于子系统而言，门面仅仅是另外一个客户端，子系统并不知道门面的存在。

三、外观的优缺点

优点：

1. 外观模式对客户屏蔽了子系统组件，从而简化了接口，减少了客户处理的对象数目并使子系统的使用更加简单。
2. 外观模式实现了子系统与客户之间的松耦合关系，而子系统内部的功能组件是紧耦合的。松耦合使得子系统的组件变化不会影响到它的客户。

缺点：

1. 如果增加新的子系统可能需要修改外观类或客户端的源代码，这样就违背了"开——闭原则"（不过这点也是不可避免）。

四、使用场景

在以下情况下可以考虑使用外观模式：

- 外一个复杂的子系统提供一个简单的接口
- 提供子系统的独立性
- 在层次化结构中，可以使用外观模式定义系统中每一层的入口。其中三层架构就是这样的一个例子。

五、总结

到这里外观模式的介绍就结束了，外观模式，为子系统的一组接口提供一个统一的接口，该模式定义了一个高层接口，这一个高层接口使的子系统更加容易使用。并且外观模式可以解决层结构分离、降低系统耦合度和为新旧系统交互提供接口功能。

本文所有源码：设计模式之外观模式

C#设计模式(12)——享元模式（Flyweight Pattern）

一、引言

在软件开发过程，如果我们需要重复使用某个对象的时候，如果我们重复地使用new创建这个对象的话，这样我们在内存就需要多次地去申请内存空间了，这样可能会出现内存使用越来越多的情况，这样的问题是非常严重，然而享元模式可以解决这个问题，下面具体看看享元模式是如何去解决这个问题的。

二、享元模式的详细介绍

在前面说了，享元模式可以解决上面的问题了，在介绍享元模式之前，让我们先要分析下如果去解决上面那个问题，上面的问题就是重复创建了同一个对象，如果让我们去解决这个问题肯定会这样想：“既然都是同一个对象，能不能只创建一个对象，然后下次需要创建这个对象的时候，让它直接用已经创建好了的对象就好了”，也就是说——让一个对象共享。不错，这个也是享元模式的实现精髓所在。

2.1 定义

介绍完享元模式的精髓之后，让我们具体看看享元模式的正式定义：

享元模式——运用共享技术有效地支持大量细粒度的对象。享元模式可以避免大量相似类的开销，在软件开发中如果需要生成大量细粒度的类实例来表示数据，如果这些实例除了几个参数外基本上都是相同的，这时候就可以使用享元模式来大幅度减少需要实例化类的数量。如果能把这些参数（指的这些类实例不同的参数）移动类实例外面，在方法调用时将他们传递进来，这样就可以通过共享大幅度地减少单个实例的数目。（这个也是享元模式的实现要领）,然而我们把类实例外面的参数称为享元对象的外部状态，把在享元对象内部定义称为内部状态。具体享元对象的内部状态与外部状态的定义为：

内部状态：在享元对象的内部并且不会随着环境的改变而改变的共享部分

外部状态：随环境改变而改变的，不可以共享的状态。

2.2 享元模式实现

分析完享元模式的实现思路之后，相信大家实现享元模式肯定没什么问题了，下面以一个实际的应用来实现下享元模式。这个例子是：一个文本编辑器中会出现很多字面，使用享元模式去实现这个文本编辑器的话，会把每个字面做成一个享元对象。享元对象的内部状态就是这个字面，而字母在文本中的位置和字体风格等其他信息就是它的外部状态。下面就以这个例子来实现下享元模式，具体实现代码如下：

```
/// <summary>
    /// 客户端调用
    /// </summary>
    class Client
    {
        static void Main(string[] args)
        {
            // 定义外部状态，例如字母的位置等信息
            int externalstate = 10;
            // 初始化享元工厂
            FlyweightFactory factory = new FlyweightFactory();

            // 判断是否已经创建了字母A，如果已经创建就直接使用创建的对象A
            Flyweight fa = factory.GetFlyweight("A");
            if (fa != null)
            {
                // 把外部状态作为享元对象的方法调用参数
                fa.Operation(--externalstate);
            }
            // 判断是否已经创建了字母B
            Flyweight fb = factory.GetFlyweight("B");
            if (fb != null)
            {
                fb.Operation(--externalstate);
            }
            // 判断是否已经创建了字母C
            Flyweight fc = factory.GetFlyweight("C");
            if (fc != null)
            {
                fc.Operation(--externalstate);
            }
            // 判断是否已经创建了字母D
            Flyweight fd= factory.GetFlyweight("D");
            if (fd != null)
            {
                fd.Operation(--externalstate);
            }
            else
            {
                Console.WriteLine("驻留池中不存在字符串D");
                // 这时候就需要创建一个对象并放入驻留池中
                ConcreteFlyweight d = new ConcreteFlyweight("D");
                factory.flyweights.Add("D", d);
            }

            Console.Read();
        }
    }

    /// <summary>
    /// 享元工厂，负责创建和管理享元对象
    /// </summary>
```

```
    public class FlyweightFactory
    {
        // 最好使用泛型Dictionary<string,Flyweighy>
        //public Dictionary<string, Flyweight> flyweights = new Dic
        public Hashtable flyweights = new Hashtable();

        public FlyweightFactory()
        {
            flyweights.Add("A", new ConcreteFlyweight("A"));
            flyweights.Add("B", new ConcreteFlyweight("B"));
            flyweights.Add("C", new ConcreteFlyweight("C"));
        }

        public Flyweight GetFlyweight(string key)
        {
```

// 更好的实现如下 //Flyweight flyweight = flyweights[key] as Flyweight; //if (flyweight == null) //{ // Console.WriteLine("驻留池中不存在字符串" + key); // flyweight = new ConcreteFlyweight(key); //} //return flyweight;

```
return flyweights[key] as Flyweight;
        }
    }

    /// <summary>
    ///  抽象享元类，提供具体享元类具有的方法
    /// </summary>
    public abstract class Flyweight
    {
        public abstract void Operation(int extrinsicstate);
    }

    // 具体的享元对象，这样我们不把每个字母设计成一个单独的类了，而是作为把共享
    public class ConcreteFlyweight : Flyweight
    {
        // 内部状态
        private string intrinsicstate ;

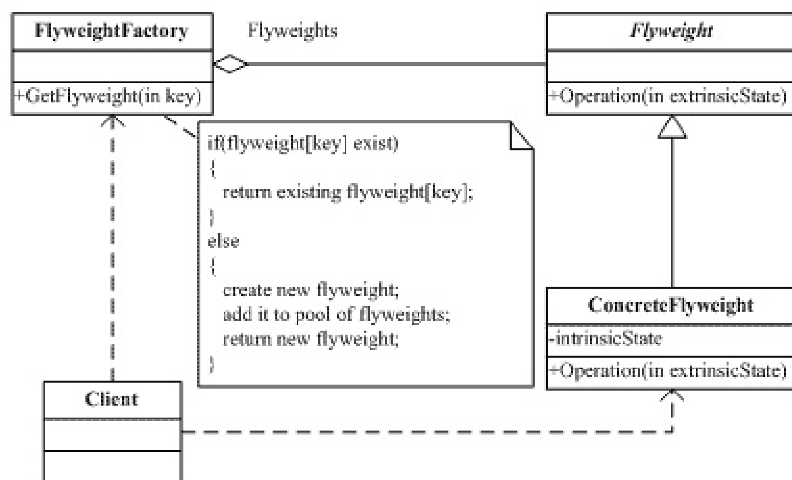
        // 构造函数
        public ConcreteFlyweight(string innerState)
        {
            this.intrinsicstate = innerState;
        }

        /// <summary>
        /// 享元类的实例方法
        /// </summary>
        /// <param name="extrinsicstate">外部状态</param>
        public override void Operation(int extrinsicstate)
        {
            Console.WriteLine("具体实现类: intrinsicstate {0}, extri
        }
    }
```

在享元模式的实现中，我们没有像之前一样，把一个细粒度的类实例设计成一个单独的类，而是把它作为共享对象的内部状态放在共享类的内部定义，具体的解释注释中都有了，大家可以参考注释去进一步理解享元模式。

2.3 享元模式的类图

看完享元模式的实现之后，为了帮助大家理清楚享元模式中各类之间的关系，下面给出上面实现代码中的类图，如下所示：



（摘自http://www.cnblogs.com/zhenyulu/articles/55793.html）

在上图中，涉及的角色如下几种角色：

抽象享元角色（Flyweight）:此角色是所有的具体享元类的基类，为这些类规定出需要实现的公共接口。那些需要外部状态的操作可以通过调用方法以参数形式传入。

具体享元角色（ConcreteFlyweight）：实现抽象享元角色所规定的接口。如果有内部状态的话，可以在类内部定义。

享元工厂角色（FlyweightFactory）：本角色复杂创建和管理享元角色。本角色必须保证享元对象可以被系统适当地共享，当一个客户端对象调用一个享元对象的时候，享元工厂角色检查系统中是否已经有一个符合要求的享元对象，如果已经存在，享元工厂角色就提供已存在的享元对象，如果系统中没有一个符合的享元对象的话，享元工厂角色就应当创建一个合适的享元对象。

客户端角色（Client）：本角色需要存储所有享元对象的外部状态。

注：上面的实现只是单纯的享元模式，同时还有复合的享元模式，由于复合享元模式较复杂，这里就不给出实现了。

三、享元模式的优缺点

分析完享元模式的实现之后，让我们继续分析下享元模式的优缺点：

优点：

1. 降低了系统中对象的数量，从而降低了系统中细粒度对象给内存带来的压力。

缺点：

1. 为了使对象可以共享，需要将一些状态外部化，这使得程序的逻辑更复杂，使系统复杂化。
2. 享元模式将享元对象的状态外部化，而读取外部状态使得运行时间稍微变长。

四、使用场景

在下面所有条件都满足时，可以考虑使用享元模式：

- 一个系统中有大量的对象；
- 这些对象耗费大量的内存；
- 这些对象中的状态大部分都可以被外部化
- 这些对象可以按照内部状态分成很多的组，当把外部对象从对象中剔除时，每一个组都可以仅用一个对象代替
- 软件系统不依赖这些对象的身份，

满足上面的条件的系统可以使用享元模式。但是使用享元模式需要额外维护一个记录子系统已有的所有享元的表，而这也需要耗费资源，所以，应当在有足够多的享元实例可共享时才值得使用享元模式。

注：在.NET类库中，string类的实现就使用了享元模式，更多内容可以参考字符串驻留池的介绍，同时也可以参考这个博文深入理解.NET中string类的设计——http://www.cnblogs.com/artech/archive/2010/11/25/internedstring.html

五、总结

到这里，享元模式的介绍就结束了，享元模式主要用来解决由于大量的细粒度对象所造成的内存开销的问题，它在实际的开发中并不常用，可以作为底层的提升性能的一种手段。

C#设计模式(13)——代理模式（Proxy Pattern）

一、引言

在软件开发过程中，有些对象有时候会由于网络或其他的障碍，以至于不能够或者不能直接访问到这些对象，如果直接访问对象给系统带来不必要的复杂性，这时候可以在客户端和目标对象之间增加一层中间层，让代理对象代替目标对象，然后客户端只需要访问代理对象，由代理对象去帮我们去请求目标对象并返回结果给客户端，这样的一个解决思路就是今天要介绍的代理模式。

二、代理模式的详细介绍

代理模式按照使用目的可以分为以下几种：

- 远程（**Remote**）代理：为一个位于不同的地址空间的对象提供一个局域代表对象。这个不同的地址空间可以是本电脑中，也可以在另一台电脑中。最典型的例子就是——客户端调用Web服务或WCF服务。
- 虚拟（**Virtual**）代理：根据需要创建一个资源消耗较大的对象，使得对象只在需要时才会被真正创建。
- **Copy-on-Write**代理：虚拟代理的一种，把复制（或者叫克隆）拖延到只有在客户端需要时，才真正采取行动。
- 保护（**Protect or Access**）代理：控制一个对象的访问，可以给不同的用户提供不同级别的使用权限。
- 防火墙（**Firewall**）代理：保护目标不让恶意用户接近。
- 智能引用（**Smart Reference**）代理：当一个对象被引用时，提供一些额外的操作，比如将对此对象调用的次数记录下来等。
- **Cache**代理：为某一个目标操作的结果提供临时的存储空间，以便多个客户端可以这些结果。

在哦上面所有种类的代理模式中，虚拟代理、远程代理、智能引用代理和保护代理较为常见的代理模式。下面让我们具体看看代理模式的具体定义。

2.1 定义

代理模式——就是给某一个对象提供一个代理，并由代理对象控制对原对象的引用。在一些情况下，一个客户不想或者不能直接引用一个对象，而代理对象可以在客户端和目标对象之间起到中介的作用。例如电脑桌面的快捷方式就是一个代理对象，快捷方式是它所引用的程序的一个代理。

2.2 代理模式实现

看完代理模式的描述之后，下面以一个生活中的例子来解释下代理模式，在现实生活中，如果有同事出国或者朋友出国的情况下，我们经常会拖这位朋友帮忙带一些电子产品或化妆品等东西，这个场景中，出国的朋友就是一个代理，他（她）是他（她）朋友的一个代理，由于他朋友不能去国外买东西，他却可以，所以朋友们都托他帮忙带一些东西的。下面就以这个场景来实现下代理模式，具体代码如下：

```
  // 客户端调用
    class Client
    {
        static void Main(string[] args)
        {
            // 创建一个代理对象并发出请求
            Person proxy = new Friend();
            proxy.BuyProduct();
            Console.Read();
        }
    }

    // 抽象主题角色
    public abstract class Person
    {
        public abstract void BuyProduct();
    }

    //真实主题角色
    public class RealBuyPerson : Person
    {
        public override void BuyProduct()
        {
            Console.WriteLine("帮我买一个IPhone和一台苹果电脑");
        }
    }

    // 代理角色
    public class Friend:Person
    {
        // 引用真实主题实例
        RealBuyPerson realSubject;

        public override void BuyProduct()
        {
            Console.WriteLine("通过代理类访问真实实体对象的方法");
            if (realSubject == null)
            {
                realSubject = new RealBuyPerson();
            }

            this.PreBuyProduct();
            // 调用真实主题方法
            realSubject.BuyProduct();
            this.PostBuyProduct();
        }
```

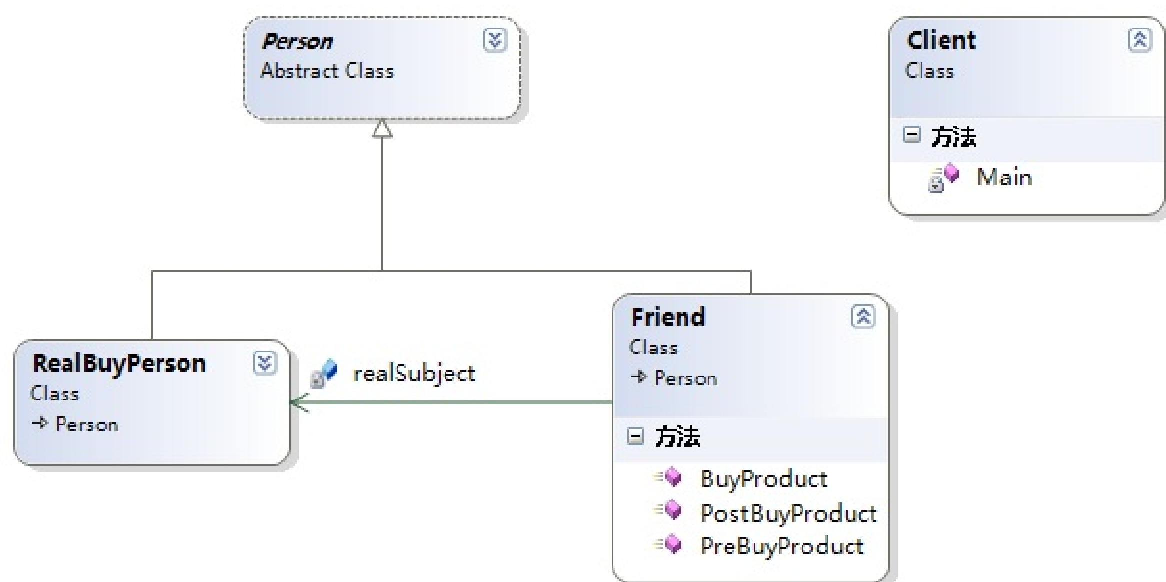
```
        // 代理角色执行的一些操作
        public void PreBuyProduct()
        {
            // 可能不知一个朋友叫这位朋友带东西，首先这位出国的朋友要对每一位
            Console.WriteLine("我怕弄糊涂了，需要列一张清单，张三：要带相机
        }

        // 买完东西之后，代理角色需要针对每位朋友需要的对买来的东西进行分类
        public void PostBuyProduct()
        {
            Console.WriteLine("终于买完了，现在要对东西分一下，相机是张三的
        }
    }
```

在上面的代码中都有相应的注释，这里也不多解释了。

2.3 代理模式的类图结构

看完代理模式的实现之后，下面就以上面的例子来分析下代理模式的类图结构。具体的类图如下所示：



在上面类图中，代理模式所涉及的角色有三个：

抽象主题角色（**Person**）：声明了真实主题和代理主题的公共接口，这样一来在使用真实主题的任何地方都可以使用代理主题。

代理主题角色（**Friend**）：代理主题角色内部含有对真实主题的引用，从而可以操作真实主题对象；代理主题角色负责在需要的时候创建真实主题对象；代理角色通常在将客户端调用传递到真实主题之前或之后，都要执行一些其他的操作，而不是单纯地将调用传递给真实主题对象。例如这里的PreBuyProduct和PostBuyProduct方法就是代理主题角色所执行的其他操作。

真实主题角色（**RealBuyPerson**）：定义了代理角色所代表的真是对象。

附：在实际开发过程中，我们在客户端添加服务引用的时候，在客户程序中会添加一些额外的类，在客户端生成的类扮演着代理主题角色，我们客户端也是直接调用这些代理角色来访问远程服务提供的操作。这个是远程代理的一个典型例子。

三、代理模式的优缺点

全面分析完代理模式之后，让我们看看这个模式的优缺点：

优点：

1. 代理模式能够将调用用于真正被调用的对象隔离，在一定程度上降低了系统的耦合度；
2. 代理对象在客户端和目标对象之间起到一个中介的作用，这样可以起到对目标对象的保护。代理对象可以在对目标对象发出请求之前进行一个额外的操作，例如权限检查等。

缺点：

1. 由于在客户端和真实主题之间增加了一个代理对象，所以会造成请求的处理速度变慢
2. 实现代理类也需要额外的工作，从而增加了系统的实现复杂度。

五、总结

到这里，代理模式的介绍就结束了，代理模式提供了对目标对象访问的代理。并且到这里，结构型模式的介绍也结束了，结构型模式包括：适配器模式、桥接模式、装饰者模式、组合模式、外观模式、享元模式和代理模式，下面开始介绍行为型模式的第一个模式：模板方法模式。

本专题所有源码：设计模式之代理模式源码

C#设计模式(14)——模板方法模式（Template Method）

一、引言

提到模板，大家肯定不免想到生活中的“简历模板”、“论文模板”、“Word中模版文件”等，在现实生活中，模板的概念就是——有一个规定的格式，然后每个人都可以根据自己的需求或情况去更新它，例如简历模板，下载下来的简历模板的格式都是相同的，然而我们下载下来简历模板之后我们可以根据自己的情况填充不同的内容要完成属于自己的简历。在设计模式中，模板方法模式中模板和生活中模板概念非常类似，下面让我们就详细介绍模板方法的定义，大家可以根据生活中模板的概念来理解模板方法的定义。

二、模板方法模式详细介绍

2.1 模板方法模式的定义

模板方法模式——在一个抽象类中定义一个操作中的算法骨架（对应于生活中的大家下载的模板），而将一些步骤延迟到子类中去实现（对应于我们根据自己的情况向模板填充内容）。模板方法使得子类可以不改变一个算法的结构前提下，重新定义算法的某些特定步骤，模板方法模式把不变行为搬到超类中，从而去除了子类中的重复代码。

2.2 模板方法模式的实现

理解了模板方法的定义之后，自然实现模板方法也不是什么难事了，下面以生活中炒蔬菜为例来实现下模板方法模式。在现实生活中，做蔬菜的步骤都大致相同，如果我们针对每种蔬菜类定义一个烧的方法，这样在每个类中都有很多相同的代码，为了解决这个问题，我们一般的思路肯定是把相同的部分抽象出来到抽象类中去定义，具体子类来实现具体的不同部分，这个思路也正式模板方法的实现精髓所在，具体实现代码如下：

```
// 客户端调用
  class Client
  {
      static void Main(string[] args)
      {
          // 创建一个菠菜实例并调用模板方法
          Spinach spinach = new Spinach();
          spinach.CookVegetabel();
          Console.Read();
```

```
        }
    }

    public abstract class Vegetabel
    {
        // 模板方法，不要把模版方法定义为Virtual或abstract方法，避免被子类重
        public  void CookVegetabel()
        {
            Console.WriteLine("抄蔬菜的一般做法");
            this.pourOil();
            this.HeatOil();
            this.pourVegetable();
            this.stir_fry();
        }

        // 第一步倒油
        public  void pourOil()
        {
            Console.WriteLine("倒油");
        }

        // 把油烧热
        public  void HeatOil()
        {
            Console.WriteLine("把油烧热");
        }

        // 油热了之后倒蔬菜下去，具体哪种蔬菜由子类决定
        public abstract void pourVegetable();

        // 开发翻炒蔬菜
        public  void stir_fry()
        {
            Console.WriteLine("翻炒");
        }
    }

    // 菠菜
    public class Spinach : Vegetabel
    {

        public override void pourVegetable()
        {
            Console.WriteLine("倒菠菜进锅中");
        }
    }

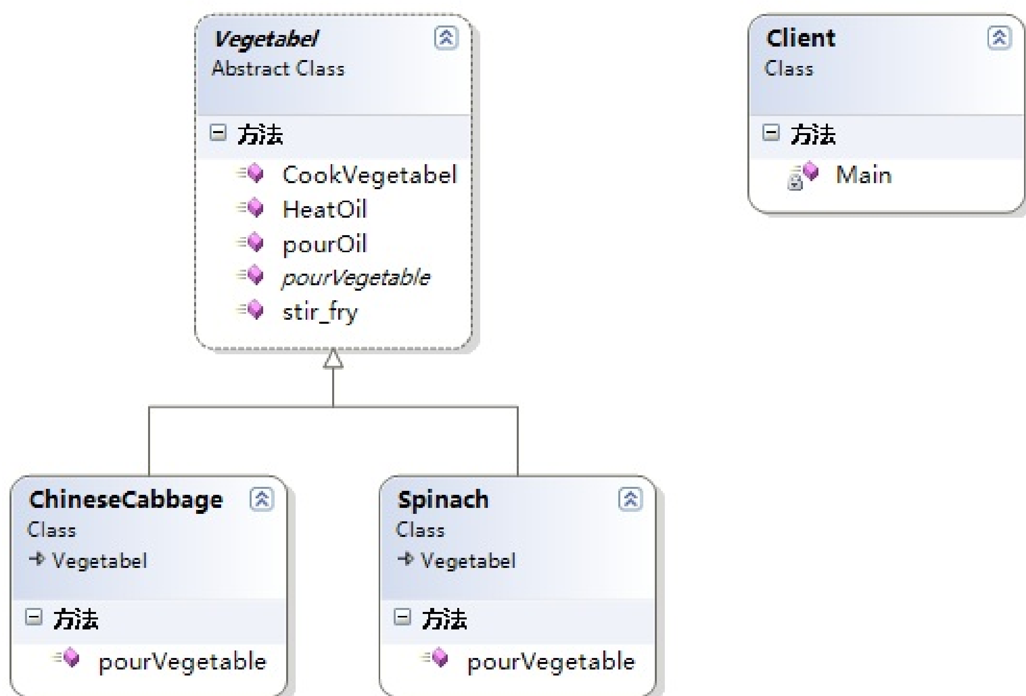
    // 大白菜
    public class ChineseCabbage : Vegetabel
    {
        public override void pourVegetable()
        {
            Console.WriteLine("倒大白菜进锅中");
```

```
        }
    }
```

在上面的实现中，具体子类中重写了导入蔬菜种类的方法，因为这个真是烧菜方法中不同的地方，所以由具体子类去实现它。

2.3 模板方法模式的类图

实现完模板方法模式之后，让我们看看模板方法的类图结构，以理清该模式中类之间的关系，具体类图如下：



模板方法模式中涉及了两个角色：

- 抽象模板角色（**Vegetable**扮演这个角色）：定义了一个或多个抽象操作，以便让子类实现，这些抽象操作称为基本操作。
- 具体模板角色（**ChineseCabbage**和**Spinach**扮演这个角色）：实现父类所定义的一个或多个抽象方法。

三、模板方法模式的优缺点

下面让我们继续分析下模板方法的优缺点。

优点：

1. 实现了代码复用
2. 能够灵活应对子步骤的变化，符合开放-封闭原则

缺点：因为引入了一个抽象类，如果具体实现过多的话，需要用户或开发人员需要花更多的时间去理清类之间的关系。

附：在.NET中模板方法的应用也很多，例如我们在开发自定义的Web控件或WinForm控件时，我们只需要重写某个控件的部分方法。

四、总结

到这里，模板方法的介绍就结束了，模板方法模式在抽象类中定义了算法的实现步骤，将这些步骤的实现延迟到具体子类中去实现，从而使所有子类复用了父类的代码，所以模板方法模式是基于继承的一种实现代码复用的技术。

C#设计模式(15)——命令模式（Command Pattern）

一、前言

之前一直在忙于工作上的事情，关于设计模式系列一直没更新，最近项目中发现，对于设计模式的了解是必不可少的，当然对于设计模式的应用那更是重要，可以说是否懂得应用设计模式在项目中是衡量一个程序员的技术水平，因为对于一个功能的实现，高级工程师和初级工程师一样都会实现，但是区别在于它们实现功能的可扩展和可维护性，也就是代码的是否“优美”、可读。但是，要更好地应用，首先就必须了解各种设计模式和其应用场景，所以我还是希望继续完成设计模式这个系列，希望通过这种总结的方式来加深自己设计模式的理解。

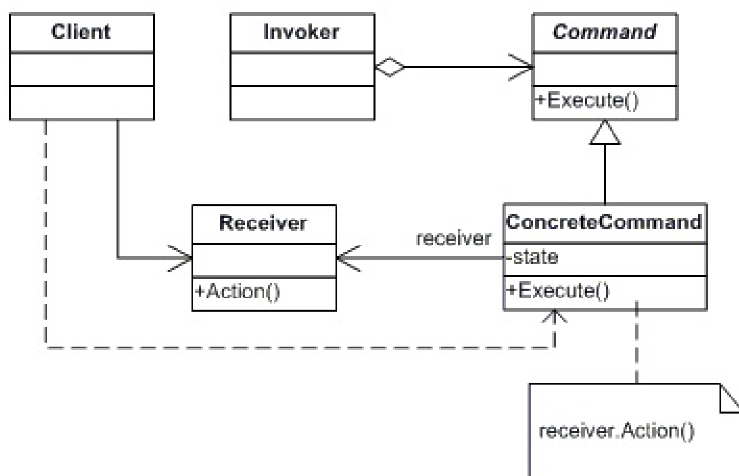
二、命令模式的介绍

2.1 命令模式的定义

命令模式属于对象的行为型模式。命令模式是把一个操作或者行为抽象为一个对象中，通过对命令的抽象化来使得发出命令的责任和执行命令的责任分隔开。命令模式的实现可以提供命令的撤销和恢复功能。

2.2 命令模式的结构

既然，命令模式是实现把发出命令的责任和执行命令的责任分割开，然而中间必须有某个对象来帮助发出命令者来传达命令，使得执行命令的接收者可以收到命令并执行命令。例如，开学了，院领导说计算机学院要进行军训，计算机学院的学生要跑1000米，院领导的话也就相当于一个命令，他不可能直接传达给到学生，他必须让教官来发出命令，并监督学生执行该命令。在这个场景中，发出命令的责任是属于学院领导，院领导充当与命令发出者的角色，执行命令的责任是属于学生，学生充当于命令接收者的角色，而教官就充当于命令的发出者或命令请求者的角色，然而命令模式的精髓就在于把每个命令抽象为对象。从而命令模式的结构如下图所示：



从命令模式的结构图可以看出，它涉及到五个角色，它们分别是：

- 客户角色：发出一个具体的命令并确定其接受者。
- 命令角色：声明了一个给所有具体命令类实现的抽象接口
- 具体命令角色：定义了一个接受者和行为的弱耦合，负责调用接受者的相应方法。
- 请求者角色：负责调用命令对象执行命令。
- 接受者角色：负责具体行为的执行。

2.3 命令模式的实现

现在，让我们以上面的军训的例子来实现一个命令模式，在实现之前，可以参考下命令模式的结构图来分析下实现过程。

军训场景中，具体的命令即是学生跑1000米，这里学生是命令的接收者，教官是命令的请求者，院领导是命令的发出者，即客户端角色。要实现命令模式，则必须需要一个抽象命令角色来声明约定，这里以抽象类来来表示。命令的传达流程是：

命令的发出者必须知道具体的命令、接受者和传达命令的请求者，对应于程序也就是在客户端角色中需要实例化三个角色的实例对象了。

命令的请求者负责调用命令对象的方法来保证命令的执行，对应于程序也就是请求者对象需要有命令对象的成员，并在请求者对象的方法内执行命令。

具体命令就是跑1000米，这自然属于学生的责任，所以是具体命令角色的成员方法，而抽象命令类定义这个命令的抽象接口。

有了上面的分析之后，具体命令模式的实现代码如下所示：

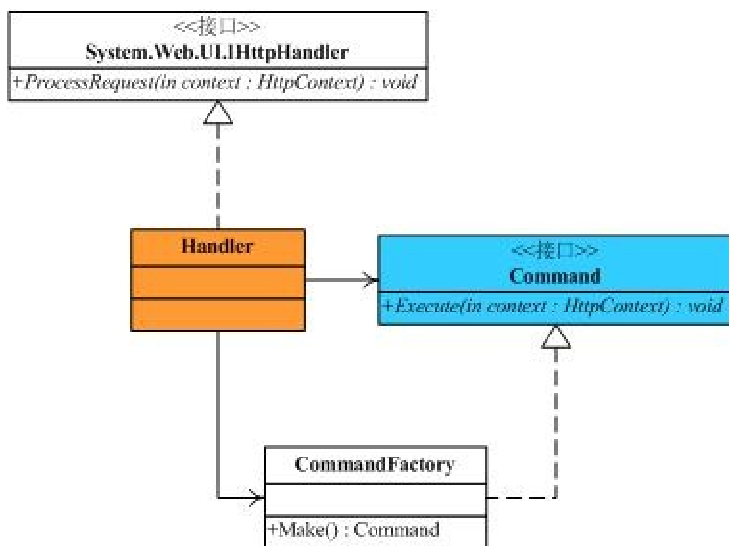
```
1
2     // 教官，负责调用命令对象执行请求
3     public class Invoke
4     {
5         public Command _command;
6
7         public Invoke(Command command)
```

```
 8          {
 9              this._command = command;
10          }
11
12          public void ExecuteCommand()
13          {
14              _command.Action();
15          }
16      }
17
18      // 命令抽象类
19      public abstract class Command
20      {
21          // 命令应该知道接收者是谁，所以有Receiver这个成员变量
22          protected Receiver _receiver;
23
24          public Command(Receiver receiver)
25          {
26              this._receiver = receiver;
27          }
28
29          // 命令执行方法
30          public abstract void Action();
31      }
32
33      //
34      public class ConcreteCommand :Command
35      {
36          public ConcreteCommand(Receiver receiver)
37              : base(receiver)
38          {
39          }
40
41          public override void Action()
42          {
43              // 调用接收的方法，因为执行命令的是学生
44              _receiver.Run1000Meters();
45          }
46      }
47
48      // 命令接收者——学生
49      public class Receiver
50      {
51          public void Run1000Meters()
52          {
53              Console.WriteLine("跑1000米");
54          }
55      }
56
57      // 院领导
58      class Program
59      {
60          static void Main(string[] args)
```

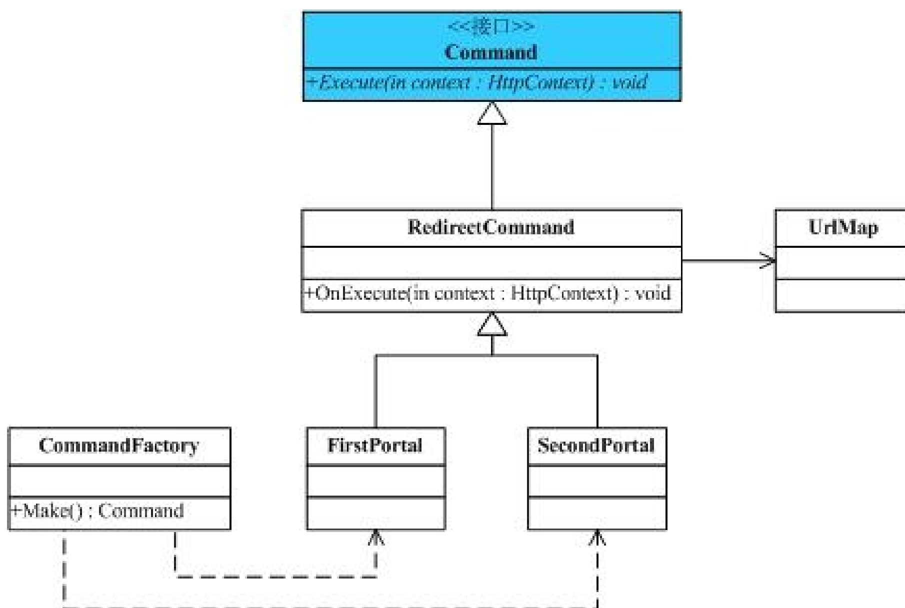
```
61          {
62              // 初始化Receiver、Invoke和Command
63              Receiver r = new Receiver();
64              Command c = new ConcreteCommand(r);
65              Invoke i = new Invoke(c);
66
67              // 院领导发出命令
68              i.ExecuteCommand();
69          }
70      }
```

三、.NET中命令模式的应用(引用TerryLee)

在ASP.NET的MVC模式中，有一种叫Front Controller的模式，它分为Handler和Command树两个部分，Handler处理所有公共的逻辑，接收HTTP Post或Get请求以及相关的参数并根据输入的参数选择正确的命令对象，然后将控制权传递到Command对象，由其完成后面的操作，这里面其实就是用到了Command模式。



Front Controller 的处理程序部分结构图



Front Controller的命令部分结构图

Handler 类负责处理各个 Web 请求，并将确定正确的 Command 对象这一职责委派给 CommandFactory 类。当 CommandFactory 返回 Command 对象后，Handler 将调用 Command 上的 Execute 方法来执行请求。具体的实现如下

```
 1 // Handler类
 2 public class Handler : IHttpHandler
 3 
 4 {
 5     public void ProcessRequest(HttpContext context)
 6 
 7     {
 8 
 9         Command command = CommandFactory.Make(context.Request.P
10 
11         command.Execute(context);
12 
13     }
14 
15     public bool IsReusable
16 
17     {
18         get
19 
20         {
21             return true;
22         }
23     }
24 }
25 
26 Command接口：
27 /// <summary>
28 /// Command
29 /// </summary>
```

```
30 public interface Command
31
32 {
33     void Execute(HttpContext context);
34 }
35
36 CommandFactory类：
37 /// <summary>
38 /// CommandFactory
39 /// </summary>
40 public class CommandFactory
41
42 {
43     public static Command Make(NameValueCollection parms)
44
45     {
46
47         string requestParm = parms["requestParm"];
48
49         Command command = null;
50
51         //根据输入参数得到不同的Command对象
52
53         switch (requestParm)
54
55         {
56             case "1":
57
58                 command = new FirstPortal();
59
60                 break;
61
62             case "2":
63
64                 command = new SecondPortal();
65
66                 break;
67
68             default:
69
70                 command = new FirstPortal();
71
72                 break;
73         }
74
75         return command;
76
77     }
78 }
79
80 RedirectCommand类：
81 public abstract class RedirectCommand : Command
82
```

```
 83 {
 84     //获得Web.Config中定义的key和url键值对，UrlMap类详见下载包中的代码
 85
 86     private UrlMap map = UrlMap.SoleInstance;
 87
 88     protected abstract void OnExecute(HttpContext context);
 89
 90     public void Execute(HttpContext context)
 91
 92     {
 93         OnExecute(context);
 94
 95         //根据key和url键值对提交到具体处理的页面
 96
 97         string url = String.Format("{0}?{1}", map.Map[context.F
 98
 99         context.Server.Transfer(url);
100
101     }
102 }
103
104 FirstPortal类：
105 public class FirstPortal : RedirectCommand
106
107 {
108     protected override void OnExecute(HttpContext context)
109
110     {
111         //在输入参数中加入项portalId以便页面处理
112
113         context.Items["portalId"] = "1";
114
115     }
116 }
117
118 SecondPortal类：
119 public class SecondPortal : RedirectCommand
120
121 {
122     protected override void OnExecute(HttpContext context)
123
124     {
125         context.Items["portalId"] = "2";
126     }
127 }
```

View Code

四、命令模式的适用场景

在下面的情况下可以考虑使用命令模式：

1. 系统需要支持命令的撤销（undo）。命令对象可以把状态存储起来，等到客户端需要撤销命令所产生的效果时，可以调用undo方法吧命令所产生的效果撤销掉。命令对象还可以提供redo方法，以供客户端在需要时，再重新实现命令效果。
2. 系统需要在不同的时间指定请求、将请求排队。一个命令对象和原先的请求发出者可以有不同的生命周期。意思为：原来请求的发出者可能已经不存在了，而命令对象本身可能仍是活动的。这时命令的接受者可以在本地，也可以在网络的另一个地址。命令对象可以串行地传送到接受者上去。
3. 如果一个系统要将系统中所有的数据消息更新到日志里，以便在系统崩溃时，可以根据日志里读回所有数据的更新命令，重新调用方法来一条一条地执行这些命令，从而恢复系统在崩溃前所做的数据更新。
4. 系统需要使用命令模式作为“CallBack(回调)”在面向对象系统中的替代。Callback即是先将一个方法注册上，然后再以后调用该方法。

五、命令模式的优缺点

命令模式使得命令发出的一个和接收的一方实现低耦合，从而有以下的优点：

- 命令模式使得新的命令很容易被加入到系统里。
- 可以设计一个命令队列来实现对请求的Undo和Redo操作。
- 可以较容易地将命令写入日志。
- 可以把命令对象聚合在一起，合成为合成命令。合成命令式合成模式的应用。

命令模式的缺点：

- 使用命令模式可能会导致系统有过多的具体命令类。这会使得命令模式在这样的系统里变得不实际。

六、总结

命令模式的实现要点在于把某个具体的命令抽象化为具体的命令类，并通过加入命令请求者角色来实现将命令发送者对命令执行者的依赖分割开，在上面军训的例子中，如果不使用命令模式的话，则命令的发送者将对命令接收者是强耦合的关系，实现代码如下：

```
1  // 院领导
2     class Program
3     {
4         static void Main(string[] args)
5         {
6            // 行为的请求者和行为的实现者之间呈现一种紧耦合关系
7             Receiver r = new Receiver();
8
9             r.Run1000Meters();
10        }
11    }
12
13    public class Receiver
14    {
15        // 操作
16        public void Run1000Meters()
17        {
18            Console.WriteLine("跑1000米");
19        }
20    }
```

到这里，本章的内容就介绍结束了，在下一章将继续为大家分享下我对迭代器模式的理解。

C#设计模式(16)——迭代器模式（Iterator Pattern）

一、引言

在上篇博文中分享了我对命令模式的理解，命令模式主要是把行为进行抽象成命令，使得请求者的行为和接受者的行为形成低耦合。在一章中，将介绍一下迭代器模式。下面废话不多说了，直接进入本博文的主题。

二、迭代器模式的介绍

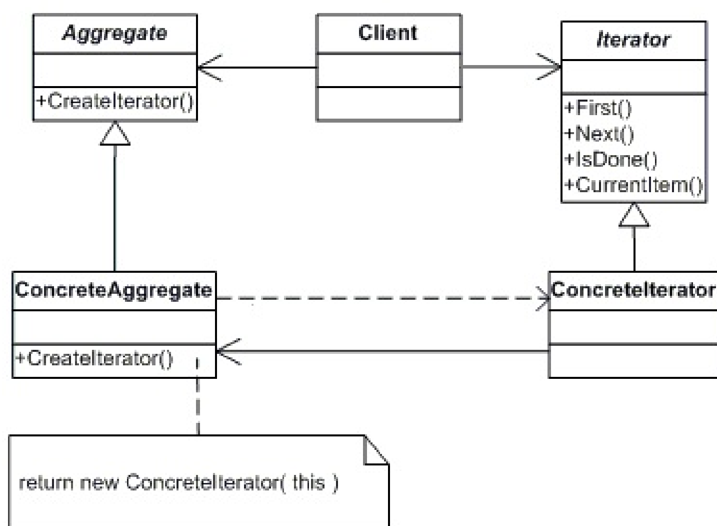
迭代器是针对集合对象而生的，对于集合对象而言，必然涉及到集合元素的添加删除操作，同时也肯定支持遍历集合元素的操作，我们此时可以把遍历操作也放在集合对象中，但这样的话，集合对象就承担太多的责任了，面向对象设计原则中有一条是单一职责原则，所以我们要尽可能地分离这些职责，用不同的类去承担不同的职责。迭代器模式就是用迭代器类来承担遍历集合元素的职责。

2.1 迭代器模式的定义

迭代器模式提供了一种方法顺序访问一个聚合对象（理解为集合对象）中各个元素，而又无需暴露该对象的内部表示，这样既可以做到不暴露集合的内部结构，又可让外部代码透明地访问集合内部的数据。

2.2 迭代器模式的结构

既然，迭代器模式承担了遍历集合对象的职责，则该模式自然存在2个类，一个是聚合类，一个是迭代器类。在面向对象涉及原则中还有一条是针对接口编程，所以，在迭代器模式中，抽象了2个接口，一个是聚合接口，另一个是迭代器接口，这样迭代器模式中就四个角色了，具体的类图如下所示：



从上图可以看出，迭代器模式由以下角色组成：

- 迭代器角色（Iterator）：迭代器角色负责定义访问和遍历元素的接口
- 具体迭代器角色（Concrete Iteraror）：具体迭代器角色实现了迭代器接口，并需要记录遍历中的当前位置。
- 聚合角色（Aggregate）：聚合角色负责定义获得迭代器角色的接口
- 具体聚合角色（Concrete Aggregate）：具体聚合角色实现聚合角色接口。

2.3 迭代器模式的实现

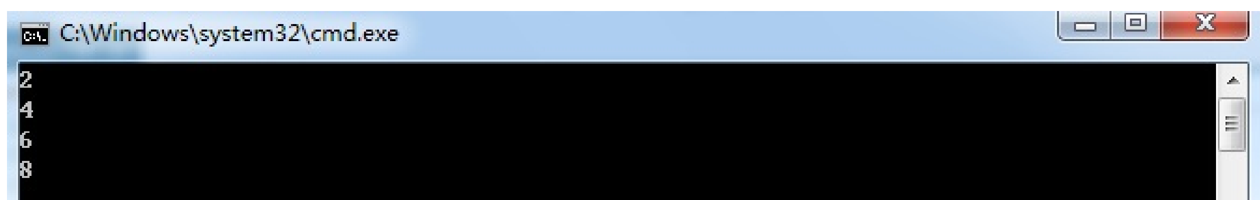
介绍完迭代器模式之后，下面就具体看看迭代器模式的实现，具体实现代码如下：

```
 1  // 抽象聚合类
 2      public interface IListCollection
 3      {
 4          Iterator GetIterator();
 5      }
 6
 7      // 迭代器抽象类
 8      public interface Iterator
 9      {
10          bool MoveNext();
11          Object GetCurrent();
12          void Next();
13          void Reset();
14      }
15
16      // 具体聚合类
17      public class ConcreteList : IListCollection
18      {
19          int[] collection;
20          public ConcreteList()
21          {
22              collection = new int[] { 2, 4, 6, 8 };
23          }
```

```
24
25          public Iterator GetIterator()
26          {
27              return new ConcreteIterator(this);
28          }
29
30          public int Length
31          {
32              get { return collection.Length; }
33          }
34
35          public int GetElement(int index)
36          {
37              return collection[index];
38          }
39      }
40
41      // 具体迭代器类
42      public class ConcreteIterator : Iterator
43      {
44          // 迭代器要集合对象进行遍历操作，自然就需要引用集合对象
45          private ConcreteList _list;
46          private int _index;
47
48          public ConcreteIterator(ConcreteList list)
49          {
50              _list = list;
51              _index = 0;
52          }
53
54
55          public bool MoveNext()
56          {
57              if (_index < _list.Length)
58              {
59                  return true;
60              }
61              return false;
62          }
63
64          public Object GetCurrent()
65          {
66              return _list.GetElement(_index);
67          }
68
69          public void Reset()
70          {
71              _index = 0;
72          }
73
74          public void Next()
75          {
76              if (_index < _list.Length)
```

```
 77                {
 78                    _index++;
 79                }
 80
 81            }
 82        }
 83
 84        // 客户端
 85        class Program
 86        {
 87            static void Main(string[] args)
 88            {
 89                Iterator iterator;
 90                IListCollection list = new ConcreteList();
 91                iterator = list.GetIterator();
 92
 93                while (iterator.MoveNext())
 94                {
 95                    int i = (int)iterator.GetCurrent();
 96                    Console.WriteLine(i.ToString());
 97                    iterator.Next();
 98                }
 99
100                Console.Read();
101            }
102        }
```

自然，上面代码的运行结果也是对集合每个元素的输出，具体运行结果如下图所示：



三、.NET中迭代器模式的应用

在.NET下，迭代器模式中的聚集接口和迭代器接口都已经存在了，其中IEnumerator接口扮演的就是迭代器角色，IEnumberable接口则扮演的就是抽象聚集的角色，只有一个GetEnumerator()方法，关于这两个接口的定义可以自行参考MSDN。在.NET 1.0中，.NET 类库中很多集合都已经实现了迭代器模式，大家可以用反编译工具Reflector来查看下mscorlib程序集下的System.Collections命名空间下的类，这里给出ArrayList的定义代码，具体实现代码可以自行用反编译工具查看，具体代码如下所示：

```
 1 public class ArrayList : IList, ICollection, IEnumerable, IClone
 2 {
 3     // Fields
 4     private const int _defaultCapacity = 4;
 5     private object[] _items;
 6     private int _size;
 7     [NonSerialized]
 8     private object _syncRoot;
 9     private int _version;
10     private static readonly object[] emptyArray;
11 
12     public virtual IEnumerator **GetEnumerator**();
13     public virtual IEnumerator **GetEnumerator**(int index, int
14 
15     // Properties
16     public virtual int Capacity { get; set; }
17     public virtual int Count { get; }
18     ..............// 更多代码请自行用反编译工具Reflector查看
19 }
```

通过查看源码你可以发现，ArrayList中迭代器的实现与我们前面给出的示例代码非常相似。然而，在.NET 2.0中，由于有了yield return关键字，实现迭代器模式就更简单了，关于迭代器的更多内容可以参考我的这篇博文。

四、迭代器模式的适用场景

在下面的情况下可以考虑使用迭代器模式：

- 系统需要访问一个聚合对象的内容而无需暴露它的内部表示。
- 系统需要支持对聚合对象的多种遍历。
- 系统需要为不同的聚合结构提供一个统一的接口。

五、迭代器模式的优缺点

由于迭代器承担了遍历集合的职责，从而有以下的优点：

- 迭代器模式使得访问一个聚合对象的内容而无需暴露它的内部表示，即迭代抽象。
- 迭代器模式为遍历不同的集合结构提供了一个统一的接口，从而支持同样的算法在不同的集合结构上进行操作

迭代器模式存在的缺陷：

- 迭代器模式在遍历的同时更改迭代器所在的集合结构会导致出现异常。所以使用foreach语句只能在对集合进行遍历，不能在遍历的同时更改集合中的元素。

六、总结

到这里，本博文的内容就介绍结束了，迭代器模式就是抽象一个迭代器类来分离了集合对象的遍历行为，这样既可以做到不暴露集合的内部结构，又可让外部代码透明地访问集合内部的数据。在一篇博文中将为大家介绍观察者模式。

C#设计模式(17)——观察者模式（Observer Pattern）

一、引言

在现实生活中，处处可见观察者模式，例如，微信中的订阅号，订阅博客和QQ微博中关注好友，这些都属于观察者模式的应用。在这一章将分享我对观察者模式的理解，废话不多说了，直接进入今天的主题。

二、 观察者模式的介绍

2.1 观察者模式的定义

从生活中的例子可以看出，只要对订阅号进行关注的客户端，如果订阅号有什么更新，就会直接推送给订阅了的用户。从中，我们就可以得出观察者模式的定义。

观察者模式定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象，这个主题对象在状态发生变化时，会通知所有观察者对象，使它们能够自动更新自己的行为。

2.2 观察者模式的结构

从上面观察者模式的定义和生活中的例子，很容易知道，观察者模式中首先会存在两个对象，一个是观察者对象，另一个就是主题对象，然而，根据面向接口编程的原则，则自然就有抽象主题角色和抽象观察者角色。理清楚了观察者模式中涉及的角色后，接下来就要理清他们之间的关联了，要想主题对象状态发生改变时，能通知到所有观察者角色，则自然主题角色必须所有观察者的引用，这样才能在自己状态改变时，通知到所有观察者。有了上面的分析，下面观察者的结构图也就很容易理解了。具体结构图如下所示：

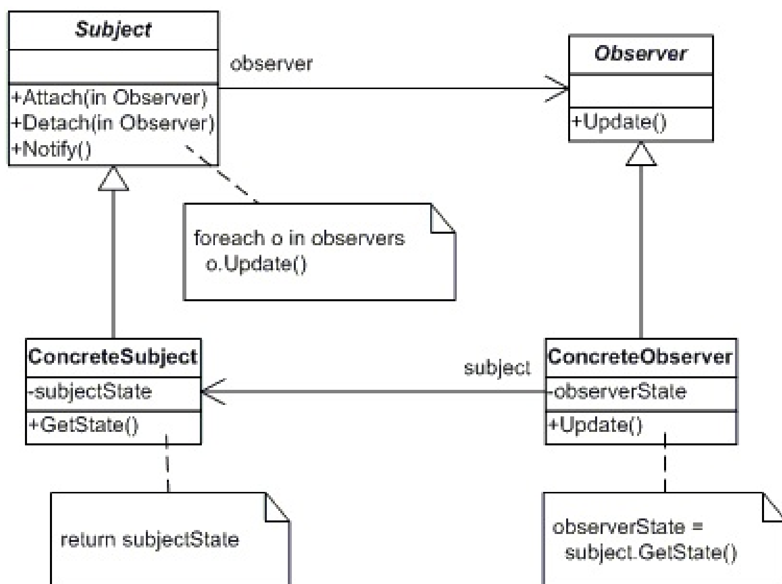


图 观察者模式结构图

可以看出，在观察者模式的结构图有以下角色：

- 抽象主题角色（Subject）：抽象主题把所有观察者对象的引用保存在一个列表中，并提供增加和删除观察者对象的操作，抽象主题角色又叫做抽象被观察者角色，一般由抽象类或接口实现。
- 抽象观察者角色（Observer）：为所有具体观察者定义一个接口，在得到主题通知时更新自己，一般由抽象类或接口实现。
- 具体主题角色（ConcreteSubject）：实现抽象主题接口，具体主题角色又叫做具体被观察者角色。
- 具体观察者角色（ConcreteObserver）：实现抽象观察者角色所要求的接口，以便使自身状态与主题的状态相协调。

2.3 观察者模式的实现

下面以微信订阅号的例子来说明观察者模式的实现。现在要实现监控腾讯游戏订阅号的状态的变化。这里一开始不采用观察者模式来实现，而通过一步步重构的方式，最终重构为观察者模式。因为一开始拿到需求，自然想到有两个类，一个是腾讯游戏订阅号类，另一个是订阅者类。订阅号类中必须引用一个订阅者对象，这样才能在订阅号状态改变时，调用这个订阅者对象的方法来通知到订阅者对象。有了这个分析，自然实现的代码如下所示：

```
 1  // 腾讯游戏订阅号类
 2     public class TenxunGame
 3     {
 4         // 订阅者对象
 5         public Subscriber Subscriber {get;set;}
 6
 7         public String Symbol {get; set;}
 8
 9         public string Info {get ;set;}
10
```

```
11          public void Update()
12          {
13              if (Subscriber != null)
14              {
15                  // 调用订阅者对象来通知订阅者
16                  Subscriber.ReceiveAndPrintData(this);
17              }
18          }
19
20      }
21
22      // 订阅者类
23      public class Subscriber
24      {
25          public string Name { get; set; }
26          public Subscriber(string name)
27          {
28              this.Name = name;
29          }
30
31          public void ReceiveAndPrintData(TenxunGame txGame)
32          {
33              Console.WriteLine("Notified {0} of {1}'s" + " Info i
34          }
35      }
36
37      // 客户端测试
38      class Program
39      {
40          static void Main(string[] args)
41          {
42              // 实例化订阅者和订阅号对象
43              Subscriber LearningHardSub = new Subscriber("Learnin
44              TenxunGame txGame = new TenxunGame();
45
46              txGame.Subscriber = LearningHardSub;
47              txGame.Symbol = "TenXun Game";
48              txGame.Info = "Have a new game published ....";
49
50              txGame.Update();
51
52              Console.ReadLine();
53          }
54      }
```

上面代码确实实现了监控订阅号的任务。但这里的实现存在下面几个问题：

- TenxunGame类和Subscriber类之间形成了一种双向依赖关系，即TenxunGame调用了Subscriber的ReceiveAndPrintData方法，而Subscriber调用了TenxunGame类的属性。这样的实现，如果有其中一个类变化将引起另一个类的改变。

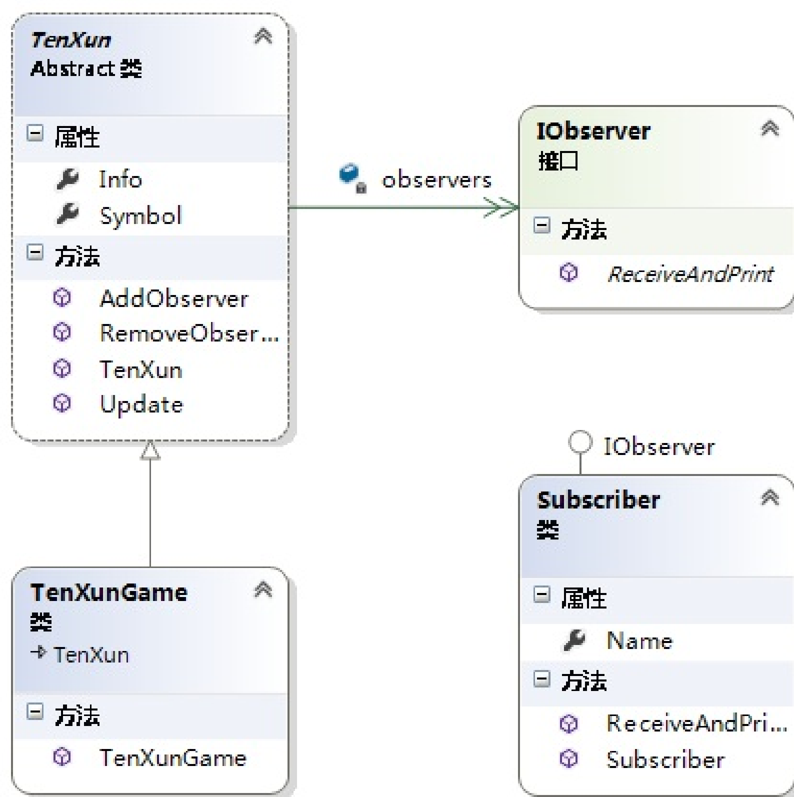
- 当出现一个新的订阅者时，此时不得不修改TenxunGame代码，即添加另一个订阅者的引用和在Update方法中调用另一个订阅者的方法。

上面的设计违背了“开放——封闭”原则，显然，这不是我们想要的。对此我们要做进一步的抽象，既然这里变化的部分是新订阅者的出现，这样我们可以对订阅者抽象出一个接口，用它来取消TenxunGame类与具体的订阅者之间的依赖，做这样一步改进，确实可以解决TenxunGame类与具体订阅者之间的依赖，使其依赖与接口，从而形成弱引用关系，但还是不能解决出现一个订阅者不得不修改TenxunGame代码的问题。对此，我们可以做这样的思考——订阅号存在多个订阅者，我们可以采用一个列表来保存所有的订阅者对象，在订阅号内部再添加对该列表的操作，这样不就解决了出现新订阅者的问题了嘛。并且订阅号也属于变化的部分，所以，我们可以采用相同的方式对订阅号进行抽象，抽象出一个抽象的订阅号类，这样也就可以完美解决上面代码存在的问题了，具体的实现代码为：

```
 1 // 订阅号抽象类
 2     public abstract class TenXun
 3     {
 4         // 保存订阅者列表
 5         private List<IObserver> observers = new List<IObserver>(
 6 
 7         public string Symbol { get; set; }
 8         public string Info { get; set; }
 9         public TenXun(string symbol, string info)
10         {
11             this.Symbol = symbol;
12             this.Info = info;
13         }
14 
15         #region 新增对订阅号列表的维护操作
16         public void AddObserver(IObserver ob)
17         {
18             observers.Add(ob);
19         }
20         public void RemoveObserver(IObserver ob)
21         {
22             observers.Remove(ob);
23         }
24         #endregion
25 
26         public void Update()
27         {
28             // 遍历订阅者列表进行通知
29             foreach (IObserver ob in observers)
30             {
31                 if (ob != null)
32                 {
33                     ob.ReceiveAndPrint(this);
34                 }
35             }
36         }
37     }
```

```
38
39      // 具体订阅号类
40      public class TenXunGame : TenXun
41      {
42          public TenXunGame(string symbol, string info)
43              : base(symbol, info)
44          {
45          }
46      }
47
48      // 订阅者接口
49      public interface IObserver
50      {
51          void ReceiveAndPrint(TenXun tenxun);
52      }
53
54      // 具体的订阅者类
55      public class Subscriber : IObserver
56      {
57          public string Name { get; set; }
58          public Subscriber(string name)
59          {
60              this.Name = name;
61          }
62
63          public void ReceiveAndPrint(TenXun tenxun)
64          {
65              Console.WriteLine("Notified {0} of {1}'s" + " Info i
66          }
67      }
68
69      // 客户端测试
70      class Program
71      {
72          static void Main(string[] args)
73          {
74              TenXun tenXun = new TenXunGame("TenXun Game", "Have
75
76              // 添加订阅者
77              tenXun.AddObserver(new Subscriber("Learning Hard"));
78              tenXun.AddObserver(new Subscriber("Tom"));
79
80              tenXun.Update();
81
82              Console.ReadLine();
83          }
84      }
```

上面代码是我们进行重构后的实现，重构后的代码实现类图如下所示：



从上图可以发现，这样的实现就是观察者模式的实现。这样，在任何时候，只要调用了TenXun类的Update方法，它就会通知所有的观察者对象，同时，可以看到，观察者模式，取消了直接依赖，变为间接依赖，这样大大提供了系统的可维护性和可扩展性。这里并不是直接给出观察者模式的实现，而是通过一步步重构的方式来引出观察者模式的实现，相信通过这个方式，大家可以更深刻地理解观察者模式所解决的问题和带来的好处。

三、.NET 中观察者模式的应用

在.NET中，我们可以使用委托与事件来简化观察者模式的实现，上面的例子用事件和委托的实现如下代码所示：

```
 1 namespace ObserverInNET
 2 {
 3     class Program
 4     {
 5         // 委托充当订阅者接口类
 6         public delegate void NotifyEventHandler(object sender);
 7
 8         // 抽象订阅号类
 9         public class TenXun
10         {
11             public NotifyEventHandler NotifyEvent;
12
13             public string Symbol { get; set; }
14             public string Info { get; set; }
```

```
15              public TenXun(string symbol, string info)
16              {
17                  this.Symbol = symbol;
18                  this.Info = info;
19              }
20
21              #region 新增对订阅号列表的维护操作
22              public void AddObserver(NotifyEventHandler ob)
23              {
24                  NotifyEvent += ob;
25              }
26              public void RemoveObserver(NotifyEventHandler ob)
27              {
28                  NotifyEvent -= ob;
29              }
30
31              #endregion
32
33              public void Update()
34              {
35                  if (NotifyEvent != null)
36                  {
37                      NotifyEvent(this);
38                  }
39              }
40          }
41
42          // 具体订阅号类
43          public class TenXunGame : TenXun
44          {
45              public TenXunGame(string symbol, string info)
46                  : base(symbol, info)
47              {
48              }
49          }
50
51          // 具体订阅者类
52          public class Subscriber
53          {
54              public string Name { get; set; }
55              public Subscriber(string name)
56              {
57                  this.Name = name;
58              }
59
60              public void ReceiveAndPrint(Object obj)
61              {
62                  TenXun tenxun = obj as TenXun;
63
64                  if (tenxun != null)
65                  {
66                      Console.WriteLine("Notified {0} of {1}'s" +
67                  }
```

```
68                }
69            }
70
71            static void Main(string[] args)
72            {
73                TenXun tenXun = new TenXunGame("TenXun Game", "Have
74                Subscriber lh =  new Subscriber("Learning Hard");
75                Subscriber tom =  new Subscriber("Tom");
76
77                // 添加订阅者
78                tenXun.AddObserver(new NotifyEventHandler(lh.Receive
79                tenXun.AddObserver(new NotifyEventHandler(tom.Receiv
80
81                tenXun.Update();
82
83                Console.WriteLine("-----------------------------------
84                Console.WriteLine("移除Tom订阅者");
85                tenXun.RemoveObserver(new NotifyEventHandler(tom.Rec
86                tenXun.Update();
87
88                Console.ReadLine();
89            }
90        }
91 }
```

从上面代码可以看出，使用事件和委托实现的观察者模式中，减少了订阅者接口类的定义，此时，.NET中的委托正式充到订阅者接口类的角色。使用委托和事件，确实简化了观察者模式的实现，减少了一个IObserver接口的定义，上面代码的运行结果如下图所示：

```
file:///F:/Study/C#/博客园中例子/C#设计模式/观察者模式/ObserverInNET/bin/Debug/Observ...
Notified Learning Hard of TenXun Game's Info is: Have a new game published ....
Notified Tom of TenXun Game's Info is: Have a new game published ....
-----------------------------------
移除Tom订阅者
Notified Learning Hard of TenXun Game's Info is: Have a new game published ....
```

四、观察者模式的适用场景

在下面的情况下可以考虑使用观察者模式：

- 当一个抽象模型有两个方面，其中一个方面依赖于另一个方面，将这两者封装在独立的对象中以使它们可以各自独立地改变和复用的情况下。从方面的这个词中可以想到，观察者模式肯定在AOP（面向方面编程）中有所体现，更多内容参考：Observern Pattern in AOP.
- 当对一个对象的改变需要同时改变其他对象，而又不知道具体有多少对象有待改变的情况下。
- 当一个对象必须通知其他对象，而又不能假定其他对象是谁的情况下。

五、观察者模式的优缺点

观察者模式有以下几个优点：

- 观察者模式实现了表示层和数据逻辑层的分离，并定义了稳定的更新消息传递机制，并抽象了更新接口，使得可以有各种各样不同的表示层，即观察者。
- 观察者模式在被观察者和观察者之间建立了一个抽象的耦合，被观察者并不知道任何一个具体的观察者，只是保存着抽象观察者的列表，每个具体观察者都符合一个抽象观察者的接口。
- 观察者模式支持广播通信。被观察者会向所有的注册过的观察者发出通知。

观察者也存在以下一些缺点：

- 如果一个被观察者有很多直接和间接的观察者时，将所有的观察者都通知到会花费很多时间。
- 虽然观察者模式可以随时使观察者知道所观察的对象发送了变化，但是观察者模式没有相应的机制使观察者知道所观察的对象是怎样发生变化的。
- 如果在被观察者之间有循环依赖的话，被观察者会触发它们之间进行循环调用，导致系统崩溃，在使用观察者模式应特别注意这点。

六 总结

到这里，观察者模式的分享就介绍了。观察者模式定义了一种一对多的依赖关系，让多个观察者对象可以同时监听某一个主题对象，这个主题对象在发生状态变化时，会通知所有观察者对象，使它们能够自动更新自己，解决的是“当一个对象的改变需要同时改变多个其他对象”的问题。大家可以以微信订阅号的例子来理解观察者模式。

C#设计模式(18)——中介者模式（Mediator Pattern）

一、引言

在现实生活中，有很多中介者模式的身影，例如QQ游戏平台，聊天室、QQ群和短信平台，这些都是中介者模式在现实生活中的应用，下面就具体分享下我对中介者模式的理解。

二、 中介者模式的介绍

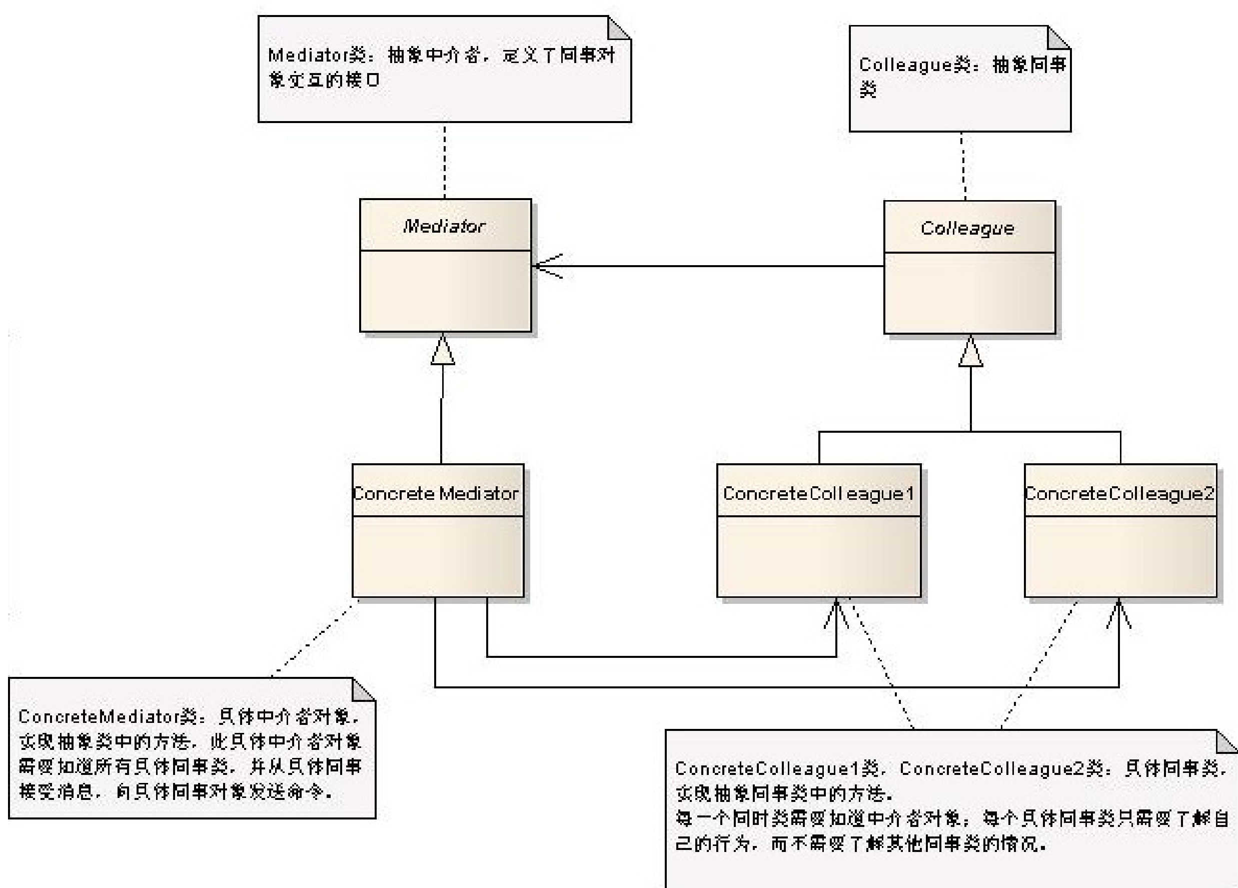
2.1 中介者模式的定义

从生活中的例子可以看出，不论是QQ游戏还是QQ群，它们都是充当一个中间平台，QQ用户可以登录这个中间平台与其他QQ用户进行交流，如果没有这些中间平台，我们如果想与朋友进行聊天的话，可能就需要当面才可以了。电话、短信也同样是一个中间平台，有了这个中间平台，每个用户都不要直接依赖与其他用户，只需要依赖这个中间平台就可以了，一切操作都由中间平台去分发。了解完中介模式在生活中的模型后，下面给出中介模式的正式定义。

中介者模式，定义了一个中介对象来封装一系列对象之间的交互关系。中介者使各个对象之间不需要显式地相互引用，从而使耦合性降低，而且可以独立地改变它们之间的交互行为。

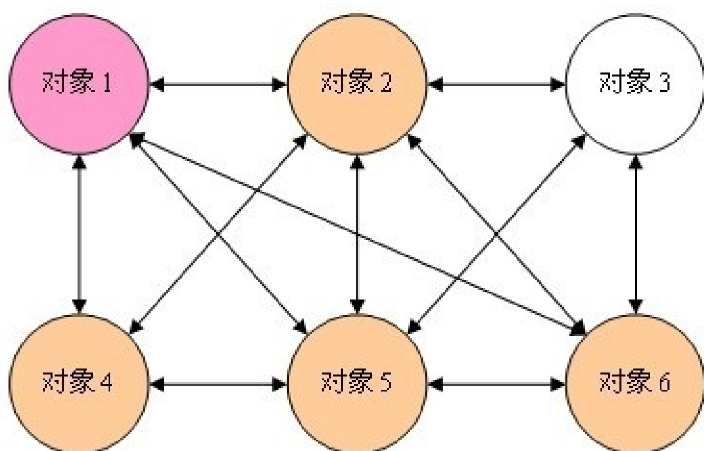
2.2 中介者模式的结构

从生活中例子自然知道，中介者模式设计两个具体对象，一个是用户类，另一个是中介者类，根据针对接口编程原则，则需要把这两类角色进行抽象，所以中介者模式中就有了4类角色，它们分别是：抽象中介者角色，具体中介者角色、抽象同事类和具体同事类。中介者类是起到协调各个对象的作用，则抽象中介者角色中则需要保存各个对象的引用。有了上面的分析，则就不难理解中介者模式的结构图了，具体结构图如下所示：

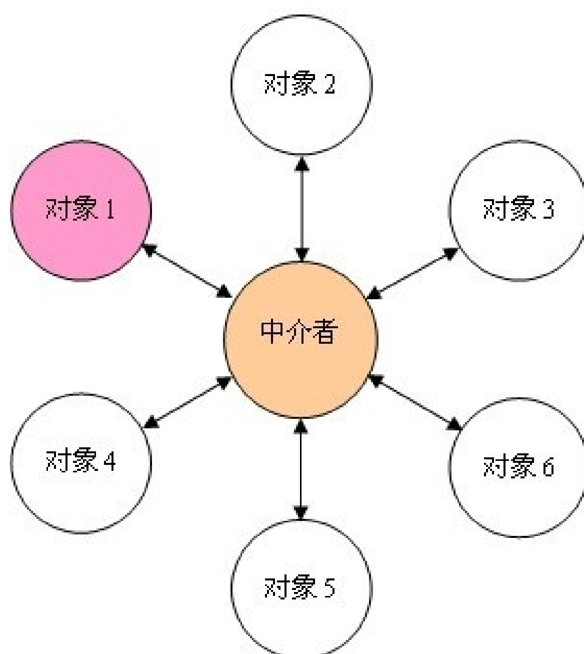


为什么要使用中介者模式

在现实生活中，中介者的存在是不可缺少的，如果没有了中介者，我们就不能与远方的朋友进行交流了。而在软件设计领域，为什么要使用中介者模式呢？如果不使用中介者模式的话，各个同事对象将会相互进行引用，如果每个对象都与多个对象进行交互时，将会形成如下图所示的网状结构。



从上图可以发现，如果不使用中介者模式的话，每个对象之间过度耦合，这样的既不利于类的复用也不利于扩展。如果引入了中介者模式，那么对象之间的关系将变成星型结构，采用中介者模式之后会形成如下图所示的结构：



从上图可以发现，使用中介者模式之后，任何一个类的变化，只会影响中介者和类本身，不像之前的设计，任何一个类的变化都会引起其关联所有类的变化。这样的设计大大减少了系统的耦合度。

2.3 中介者模式的实现

介绍完中介者模式的定义和存在的必要性后，下面就以现实生活中打牌的例子来实现下中介者模式。在现实生活中，两个人打牌，如果某个人赢了都会影响到对方状态的改变。如果此时不采用中介者模式实现的话，则上面的场景的实现如下所示：

```
 1 // 抽象牌友类
 2     public abstract class AbstractCardPartner
 3     {
 4         public int MoneyCount { get; set; }
 5 
 6         public AbstractCardPartner()
 7         {
 8             MoneyCount = 0;
 9         }
10 
11         public abstract void ChangeCount(int Count, AbstractCard
12     }
13 
14     // 牌友A类
15     public class ParterA : AbstractCardPartner
16     {
17         public override void ChangeCount(int Count, AbstractCard
18         {
19             this.MoneyCount += Count;
20             other.MoneyCount -= Count;
21         }
```

```
22     }
23
24     // 牌友B类
25     public class ParterB : AbstractCardPartner
26     {
27         public override void ChangeCount(int Count, AbstractCard
28         {
29             this.MoneyCount += Count;
30             other.MoneyCount -= Count;
31         }
32     }
33
34     class Program
35     {
36         // A,B两个人打牌
37         static void Main(string[] args)
38         {
39             AbstractCardPartner A = new ParterA();
40             A.MoneyCount = 20;
41             AbstractCardPartner B = new ParterB();
42             B.MoneyCount = 20;
43
44             // A 赢了则B的钱就减少
45             A.ChangeCount(5, B);
46             Console.WriteLine("A 现在的钱是：{0}", A.MoneyCount);/
47             Console.WriteLine("B 现在的钱是：{0}", B.MoneyCount);
48
49             // B赢了A的钱也减少
50             B.ChangeCount(10, A);
51             Console.WriteLine("A 现在的钱是：{0}", A.MoneyCount);
52             Console.WriteLine("B 现在的钱是：{0}", B.MoneyCount);
53             Console.Read();
54         }
55     }
```

上面确实完美解决了上面场景中的问题，并且使用了抽象类使具体牌友A和牌友B都依赖于抽象类，从而降低了同事类之间的耦合度。但是这样的设计，如果其中牌友A发生变化时，此时就会影响到牌友B的状态，如果涉及的对象变多的话，这时候某一个牌友的变化将会影响到其他所有相关联的牌友状态。例如牌友A算错了钱，这时候牌友A和牌友B的钱数都不正确了，如果是多个人打牌的话，影响的对象就会更多。这时候就会思考——能不能把算钱的任务交给程序或者算数好的人去计算呢，这时候就有了我们QQ游戏中的欢乐斗地主等牌类游戏了。所以上面的设计，我们还是有进一步完善的方案的，即加入一个中介者对象来协调各个对象之间的关联，这也就是中介者模式的应用了，具体完善后的实现代码如下所示：

```
1 namespace MediatorPattern
2 {
3     // 抽象牌友类
4     public abstract class AbstractCardPartner
5     {
```

```
 6          public int MoneyCount { get; set; }
 7
 8          public AbstractCardPartner()
 9          {
10              MoneyCount = 0;
11          }
12
13          public abstract void ChangeCount(int Count, AbstractMedi
14      }
15
16      // 牌友A类
17      public class ParterA : AbstractCardPartner
18      {
19          // 依赖与抽象中介者对象
20          public override void ChangeCount(int Count, AbstractMedi
21          {
22              mediator.AWin(Count);
23          }
24      }
25
26      // 牌友B类
27      public class ParterB : AbstractCardPartner
28      {
29          // 依赖与抽象中介者对象
30          public override void ChangeCount(int Count, AbstractMedi
31          {
32              mediator.BWin(Count);
33          }
34      }
35
36      // 抽象中介者类
37      public abstract class AbstractMediator
38      {
39          protected AbstractCardPartner A;
40          protected AbstractCardPartner B;
41          public AbstractMediator(AbstractCardPartner a, AbstractC
42          {
43              A = a;
44              B = b;
45          }
46
47          public abstract void AWin(int count);
48          public abstract void BWin(int count);
49      }
50
51      // 具体中介者类
52      public class MediatorPater : AbstractMediator
53      {
54          public MediatorPater(AbstractCardPartner a, AbstractCard
55              : base(a, b)
56          {
57          }
58
```

```
59          public override void AWin(int count)
60          {
61              A.MoneyCount += count;
62              B.MoneyCount -= count;
63          }
64
65          public override void BWin(int count)
66          {
67              B.MoneyCount += count;
68              A.MoneyCount -= count;
69          }
70      }
71
72      class Program
73      {
74          static void Main(string[] args)
75          {
76              AbstractCardPartner A = new ParterA();
77              AbstractCardPartner B = new ParterB();
78              // 初始钱
79              A.MoneyCount = 20;
80              B.MoneyCount = 20;
81
82              AbstractMediator mediator = new MediatorPater(A, B);
83
84              // A赢了
85              A.ChangeCount(5, mediator);
86              Console.WriteLine("A 现在的钱是：{0}", A.MoneyCount);
87              Console.WriteLine("B 现在的钱是：{0}", B.MoneyCount);
88
89              // B 赢了
90              B.ChangeCount(10, mediator);
91              Console.WriteLine("A 现在的钱是：{0}", A.MoneyCount);
92              Console.WriteLine("B 现在的钱是：{0}", B.MoneyCount);
93              Console.Read();
94          }
95      }
96  }
```

从上面实现代码可以看出，此时牌友A和牌友B都依赖于抽象的中介者类，这样如果其中某个牌友类变化只会影响到，只会影响到该变化牌友类本身和中介者类，从而解决前面实现代码出现的问题，具体的运行结果和前面实现结果一样，运行结果如下图所示：

```
file:///F:/Study/C#/博客园中例子/C#设计模式/中介者模式/DonotUseMediatorPattern/bin/De...
A 现在的钱是: 25
B 现在的钱是: 15
A 现在的钱是: 15
B 现在的钱是: 25
```

在上面的实现代码中，抽象中介者类保存了两个抽象牌友类，如果新添加一个牌友类似时，此时就不得不去更改这个抽象中介者类。可以结合观察者模式来解决这个问题，即抽象中介者对象保存抽象牌友的类别，然后添加Register和UnRegister方法来对该列表进行管理，然后在具体中介者类中修改AWin和BWin方法，遍历列表，改变自己和其他牌友的钱数。这样的设计还是存在一个问题——即增加一个新牌友时，此时虽然解决了抽象中介者类不需要修改的问题，但此时还是不得不去修改具体中介者类，即添加CWin方法，我们可以采用状态模式来解决这个问题，关于状态模式的介绍将会在下一专题进行分享。

三、中介者模式的适用场景

一般在以下情况下可以考虑使用中介者模式：

- 一组定义良好的对象，现在要进行复杂的相互通信。
- 想通过一个中间类来封装多个类中的行为，而又不想生成太多的子类。

四、中介者模式的优缺点

中介者模式具有以下几点优点：

- 简化了对象之间的关系，将系统的各个对象之间的相互关系进行封装，将各个同事类解耦，使得系统变为松耦合。
- 提供系统的灵活性，使得各个同事对象独立而易于复用。

然而，中介者模式也存在对应的缺点：

- 中介者模式中，中介者角色承担了较多的责任，所以一旦这个中介者对象出现了问题，整个系统将会受到重大的影响。例如，QQ游戏中计算欢乐豆的程序出错了，这样会造成重大的影响。
- 新增加一个同事类时，不得不去修改抽象中介者类和具体中介者类，此时可以使用观察者模式和状态模式来解决这个问题。

C#设计模式(19)——状态者模式（State Pattern）

一、引言

在上一篇文章介绍到可以使用状态者模式和观察者模式来解决中介者模式存在的问题，在本文中将首先通过一个银行账户的例子来解释状态者模式，通过这个例子使大家可以对状态者模式有一个清楚的认识，接着，再使用状态者模式来解决上一篇文章中提出的问题。

二、状态者模式的介绍

每个对象都有其对应的状态，而每个状态又对应一些相应的行为，如果某个对象有多个状态时，那么就会对应很多的行为。那么对这些状态的判断和根据状态完成的行为，就会导致多重条件语句，并且如果添加一种新的状态时，需要更改之前现有的代码。这样的设计显然违背了开闭原则。状态模式正是用来解决这样的问题的。状态模式将每种状态对应的行为抽象出来成为单独新的对象，这样状态的变化不再依赖于对象内部的行为。

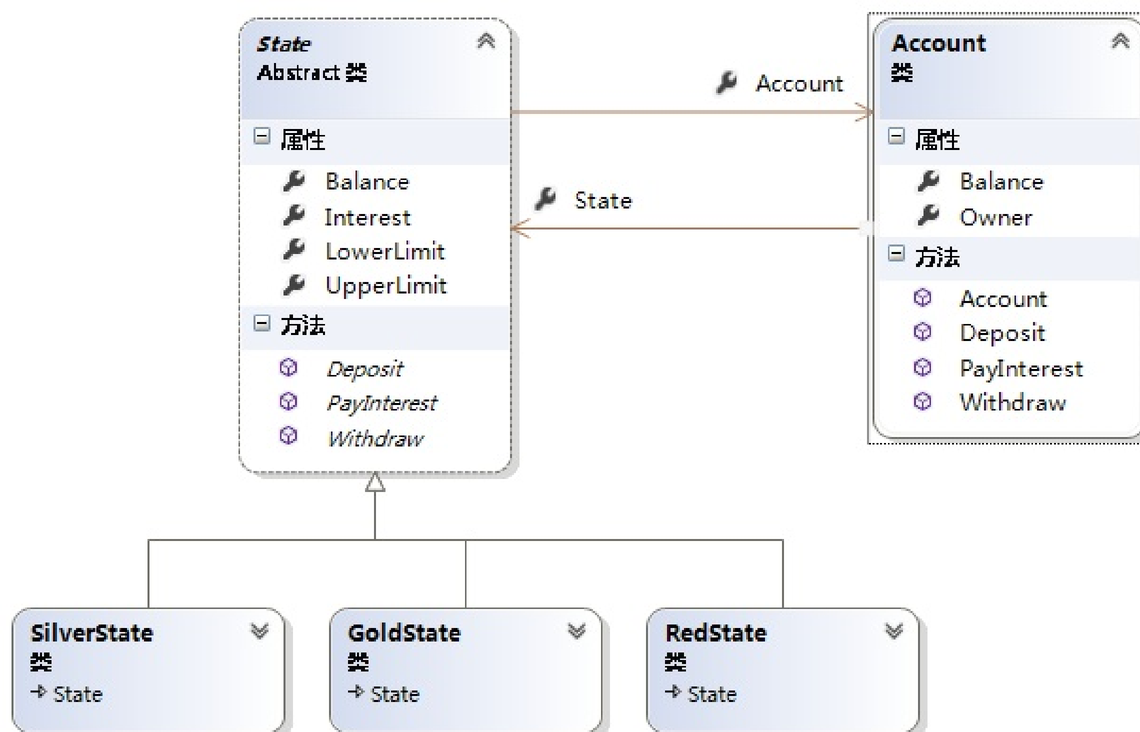
2.1 状态者模式的定义

上面对状态模式做了一个简单的介绍，这里给出状态模式的定义。

状态模式——允许一个对象在其内部状态改变时自动改变其行为，对象看起来就像是改变了它的类。

2.2 状态者模式的结构

既然状态者模式是对已有对象的状态进行抽象，则自然就有抽象状态者类和具体状态者类，而原来已有对象需要保存抽象状态者类的引用，通过调用抽象状态者的行为来改变已有对象的行为。经过上面的分析，状态者模式的结构图也就很容易理解了，具体结构图如下图示。



从上图可知，状态者模式涉及以下三个角色：

- Account类：维护一个State类的一个实例，该实例标识着当前对象的状态。
- State类：抽象状态类，定义了一个具体状态类需要实现的行为约定。
- SilveStater、GoldState和RedState类：具体状态类，实现抽象状态类的每个行为。

2.3 状态者模式的实现

下面，就以银行账户的状态来实现下状态者模式。银行账户根据余额可分为RedState、SilverState和GoldState。这些状态分别代表透支账号，新开账户和标准账户。账号余额在【-100.0，0.0】范围表示处于RedState状态，账号余额在【0.0， 1000.0】范围表示处于SilverState，账号在【1000.0， 100000.0】范围表示处于GoldState状态。下面以这样的一个场景实现下状态者模式，具体实现代码如下所示：

```
 1 namespace StatePatternSample
 2 {
 3     public class Account
 4     {
 5         public State State {get;set;}
 6         public string Owner { get; set; }
 7         public Account(string owner)
 8         {
 9             this.Owner = owner;
10             this.State = new SilverState(0.0, this);
11         }
12 
```

```
13          public double Balance { get {return State.Balance; }} /
14          // 存钱
15          public void Deposit(double amount)
16          {
17              State.Deposit(amount);
18              Console.WriteLine("存款金额为 {0:C}——", amount);
19              Console.WriteLine("账户余额为 =:{0:C}", this.Balance
20              Console.WriteLine("账户状态为: {0}", this.State.GetT
21              Console.WriteLine();
22          }
23
24          // 取钱
25          public void Withdraw(double amount)
26          {
27              State.Withdraw(amount);
28               Console.WriteLine("取款金额为 {0:C}——",amount);
29              Console.WriteLine("账户余额为 =:{0:C}", this.Balance
30              Console.WriteLine("账户状态为: {0}", this.State.GetT
31              Console.WriteLine();
32          }
33
34          // 获得利息
35          public void PayInterest()
36          {
37              State.PayInterest();
38              Console.WriteLine("Interest Paid --- ");
39              Console.WriteLine("账户余额为 =:{0:C}", this.Balance
40              Console.WriteLine("账户状态为: {0}", this.State.GetT
41              Console.WriteLine();
42          }
43      }
44
45      // 抽象状态类
46      public abstract class State
47      {
48          // Properties
49          public Account Account { get; set; }
50          public double Balance { get; set; } // 余额
51          public double Interest { get; set; } // 利率
52          public double LowerLimit { get; set; } // 下限
53          public double UpperLimit { get; set; } // 上限
54
55          public abstract void Deposit(double amount); // 存款
56          public abstract void Withdraw(double amount); // 取钱
57          public abstract void PayInterest(); // 获得的利息
58      }
59
60      // Red State意味着Account透支了
61      public class RedState : State
62      {
63          public RedState(State state)
64          {
65              // Initialize
```

```
 66             this.Balance = state.Balance;
 67             this.Account = state.Account;
 68             Interest = 0.00;
 69             LowerLimit = -100.00;
 70             UpperLimit = 0.00;
 71         }
 72 
 73         // 存款
 74         public override void Deposit(double amount)
 75         {
 76             Balance += amount;
 77             StateChangeCheck();
 78         }
 79         // 取钱
 80         public override void Withdraw(double amount)
 81         {
 82             Console.WriteLine("没有钱可以取了！");
 83         }
 84 
 85         public override void PayInterest()
 86         {
 87             // 没有利息
 88         }
 89 
 90         private void StateChangeCheck()
 91         {
 92             if (Balance > UpperLimit)
 93             {
 94                 Account.State = new SilverState(this);
 95             }
 96         }
 97     }
 98 
 99     // Silver State意味着没有利息得
100     public class SilverState :State
101     {
102         public SilverState(State state)
103             : this(state.Balance, state.Account)
104         {
105         }
106 
107         public SilverState(double balance, Account account)
108         {
109             this.Balance = balance;
110             this.Account = account;
111             Interest = 0.00;
112             LowerLimit = 0.00;
113             UpperLimit = 1000.00;
114         }
115 
116         public override void Deposit(double amount)
117         {
118             Balance += amount;
```

```
119                 StateChangeCheck();
120             }
121             public override void Withdraw(double amount)
122             {
123                 Balance -= amount;
124                 StateChangeCheck();
125             }
126
127             public override void PayInterest()
128             {
129                 Balance += Interest * Balance;
130                 StateChangeCheck();
131             }
132
133             private void StateChangeCheck()
134             {
135                 if (Balance < LowerLimit)
136                 {
137                     Account.State = new RedState(this);
138                 }
139                 else if (Balance > UpperLimit)
140                 {
141                     Account.State = new GoldState(this);
142                 }
143             }
144         }
145
146         // Gold State意味着有利息状态
147         public class GoldState : State
148         {
149             public GoldState(State state)
150             {
151                 this.Balance = state.Balance;
152                 this.Account = state.Account;
153                 Interest = 0.05;
154                 LowerLimit = 1000.00;
155                 UpperLimit = 1000000.00;
156             }
157             // 存钱
158             public override void Deposit(double amount)
159             {
160                 Balance += amount;
161                 StateChangeCheck();
162             }
163             // 取钱
164             public override void Withdraw(double amount)
165             {
166                 Balance -= amount;
167                 StateChangeCheck();
168             }
169             public override void PayInterest()
170             {
171                 Balance += Interest * Balance;
```

```
172             StateChangeCheck();
173         }
174 
175         private void StateChangeCheck()
176         {
177             if (Balance < 0.0)
178             {
179                 Account.State = new RedState(this);
180             }
181             else if (Balance < LowerLimit)
182             {
183                 Account.State = new SilverState(this);
184             }
185         }
186     }
187 
188     class App
189     {
190         static void Main(string[] args)
191         {
192             // 开一个新的账户
193             Account account = new Account("Learning Hard");
194 
195             // 进行交易
196             // 存钱
197             account.Deposit(1000.0);
198             account.Deposit(200.0);
199             account.Deposit(600.0);
200 
201             // 付利息
202             account.PayInterest();
203 
204             // 取钱
205             account.Withdraw(2000.00);
206             account.Withdraw(500.00);
207 
208             // 等待用户输入
209             Console.ReadKey();
210         }
211     }
212 }
```

上面代码的运行结果如下图所示：

```
file:///F:/Study/C#/博客园中例子/C#设计模式/状态模式/StatePatternSample/bin/Debug/Stat...
账户余额为 =:￥1,000.00
账户状态为: SilverState

存款金额为 ￥200.00——
账户余额为 =:￥1,200.00
账户状态为: GoldState

存款金额为 ￥600.00——
账户余额为 =:￥1,800.00
账户状态为: GoldState

Interest Paid ---
账户余额为 =:￥1,890.00
账户状态为: GoldState

取款金额为 ￥2,000.00——
账户余额为 =:￥-110.00
账户状态为: RedState

没有钱可以取了!
取款金额为 ￥500.00——
账户余额为 =:￥-110.00
账户状态为: RedState
```

从上图可以发现，进行存取款交易，会影响到Account内部的状态，由于状态的改变，从而影响到Account类行为的改变，而且这些操作都是发生在运行时的。

三、应用状态者模式完善中介者模式方案

在上一篇博文中，我曾介绍到中介者模式存在的问题，详细的问题描述可以参考上一篇博文。下面利用观察者模式和状态者模式来完善中介者模式，具体的实现代码如下所示：

```
 1 // 抽象牌友类
 2     public abstract class AbstractCardPartner
 3     {
 4         public int MoneyCount { get; set; }
 5 
 6         public AbstractCardPartner()
 7         {
 8             MoneyCount = 0;
 9         }
10 
11         public abstract void ChangeCount(int Count, AbstractMed
12     }
13 
14     // 牌友A类
15     public class ParterA : AbstractCardPartner
16     {
17         // 依赖与抽象中介者对象
18         public override void ChangeCount(int Count, AbstractMed
19         {
```

```
20              mediator.ChangeCount(Count);
21          }
22      }
23
24      // 牌友B类
25      public class ParterB : AbstractCardPartner
26      {
27          // 依赖与抽象中介者对象
28          public override void ChangeCount(int Count, AbstractMed
29          {
30              mediator.ChangeCount(Count);
31          }
32      }
33
34      // 抽象状态类
35      public abstract class State
36      {
37          protected AbstractMediator meditor;
38          public abstract void ChangeCount(int count);
39      }
40
41      // A赢状态类
42      public class AWinState : State
43      {
44          public AWinState(AbstractMediator concretemediator)
45          {
46              this.meditor = concretemediator;
47          }
48
49          public override void ChangeCount(int count)
50          {
51              foreach (AbstractCardPartner p in meditor.list)
52              {
53                  ParterA a = p as ParterA;
54                  //
55                  if (a != null)
56                  {
57                      a.MoneyCount += count;
58                  }
59                  else
60                  {
61                      p.MoneyCount -= count;
62                  }
63              }
64          }
65      }
66
67      // B赢状态类
68      public class BWinState : State
69      {
70          public BWinState(AbstractMediator concretemediator)
71          {
72              this.meditor = concretemediator;
```

```
 73         }
 74
 75         public override void ChangeCount(int count)
 76         {
 77             foreach (AbstractCardPartner p in meditor.list)
 78             {
 79                 ParterB b = p as ParterB;
 80                 // 如果集合对象中时B对象，则对B的钱添加
 81                 if (b != null)
 82                 {
 83                     b.MoneyCount += count;
 84                 }
 85                 else
 86                 {
 87                     p.MoneyCount -= count;
 88                 }
 89             }
 90         }
 91     }
 92
 93     // 初始化状态类
 94     public class InitState : State
 95     {
 96         public InitState()
 97         {
 98             Console.WriteLine("游戏才刚刚开始,暂时还有玩家胜出");
 99         }
100
101         public override void ChangeCount(int count)
102         {
103             //
104             return;
105         }
106     }
107
108     // 抽象中介者类
109     public abstract class AbstractMediator
110     {
111         public List<AbstractCardPartner> list = new List<Abstra
112
113         public State State { get; set; }
114
115         public AbstractMediator(State state)
116         {
117             this.State = state;
118         }
119
120         public void Enter(AbstractCardPartner partner)
121         {
122             list.Add(partner);
123         }
124
125         public void Exit(AbstractCardPartner partner)
```

```
126         {
127             list.Remove(partner);
128         }
129 
130         public void ChangeCount(int count)
131         {
132             State.ChangeCount(count);
133         }
134     }
135 
136     // 具体中介者类
137     public class MediatorPater : AbstractMediator
138     {
139         public MediatorPater(State initState)
140             : base(initState)
141         { }
142     }
143 
144     class Program
145     {
146         static void Main(string[] args)
147         {
148             AbstractCardPartner A = new ParterA();
149             AbstractCardPartner B = new ParterB();
150             // 初始钱
151             A.MoneyCount = 20;
152             B.MoneyCount = 20;
153 
154             AbstractMediator mediator = new MediatorPater(new I
155 
156             // A,B玩家进入平台进行游戏
157             mediator.Enter(A);
158             mediator.Enter(B);
159 
160             // A赢了
161             mediator.State = new AWinState(mediator);
162             mediator.ChangeCount(5);
163             Console.WriteLine("A 现在的钱是：{0}", A.MoneyCount);
164             Console.WriteLine("B 现在的钱是：{0}", B.MoneyCount);
165 
166             // B 赢了
167             mediator.State = new BWinState(mediator);
168             mediator.ChangeCount(10);
169             Console.WriteLine("A 现在的钱是：{0}", A.MoneyCount);
170             Console.WriteLine("B 现在的钱是：{0}", B.MoneyCount);
171             Console.Read();
172         }
173     }
```

四、状态者模式的应用场景

在以下情况下可以考虑使用状态者模式。

- 当一个对象状态转换的条件表达式过于复杂时可以使用状态者模式。把状态的判断逻辑转移到表示不同状态的一系列类中，可以把复杂的判断逻辑简单化。
- 当一个对象行为取决于它的状态，并且它需要在运行时刻根据状态改变它的行为时，就可以考虑使用状态者模式。

五、状态者模式的优缺点

状态者模式的主要优点是：

- 将状态判断逻辑每个状态类里面，可以简化判断的逻辑。
- 当有新的状态出现时，可以通过添加新的状态类来进行扩展，扩展性好。

状态者模式的主要缺点是：

- 如果状态过多的话，会导致有非常多的状态类，加大了开销。

六、总结

状态者模式是对对象状态的抽象，从而把对象中对状态复杂的判断逻辑已到各个状态类里面，从而简化逻辑判断。在下一篇文章将分享我对策略模式的理解。

C#设计模式(20)——策略者模式（Stragety Pattern）

一、引言

前面主题介绍的状态模式是对某个对象状态的抽象，而本文要介绍的策略模式也就是对策略进行抽象，策略的意思就是方法，所以也就是对方法的抽象，下面具体分享下我对策略模式的理解。

二、策略者模式介绍

2.1 策略模式的定义

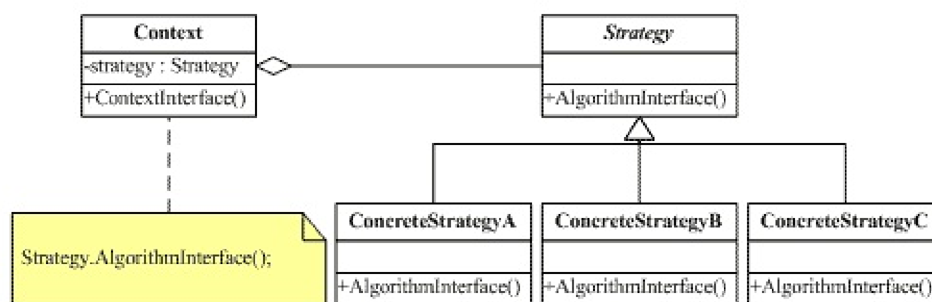
在现实生活中，策略模式的例子也非常常见，例如，中国的所得税，分为企业所得税、外商投资企业或外商企业所得税和个人所得税，针对于这3种所得税，针对每种，所计算的方式不同，个人所得税有个人所得税的计算方式，而企业所得税有其对应计算方式。如果不采用策略模式来实现这样一个需求的话，可能我们会定义一个所得税类，该类有一个属性来标识所得税的类型，并且有一个计算税收的CalculateTax()方法，在该方法体内需要对税收类型进行判断，通过if-else语句来针对不同的税收类型来计算其所得税。这样的实现确实可以解决这个场景吗，但是这样的设计不利于扩展，如果系统后期需要增加一种所得税时，此时不得不回去修改CalculateTax方法来多添加一个判断语句，这样明白违背了“开放——封闭”原则。此时，我们可以考虑使用策略模式来解决这个问题，既然税收方法是这个场景中的变化部分，此时自然可以想到对税收方法进行抽象。具体的实现代码见2.3部分。

前面介绍了策略模式用来解决的问题，下面具体给出策略的定义。策略模式是针对一组算法，将每个算法封装到具有公共接口的独立的类中，从而使它们可以相互替换。策略模式使得算法可以在不影响到客户端的情况下发生变化。

2.2 策略模式的结构

策略模式是对算法的包装，是把使用算法的责任和算法本身分割开，委派给不同的对象负责。策略模式通常把一系列的算法包装到一系列的策略类里面。用一句话概括策略模式就是——“将每个算法封装到不同的策略类中，使得它们可以互换”。

下面是策略模式的结构图：



该模式涉及到三个角色：

- 环境角色（Context）：持有一个Strategy类的引用
- 抽象策略角色（Strategy）：这是一个抽象角色，通常由一个接口或抽象类来实现。此角色给出所有具体策略类所需实现的接口。
- 具体策略角色（ConcreteStrategy）：包装了相关算法或行为。

2.3 策略模式的实现

下面就以所得税的例子来实现下策略模式，具体实现代码如下所示：

```
 1 namespace StrategyPattern
 2 {
 3     // 所得税计算策略
 4     public interface ITaxStragety
 5     {
 6         double CalculateTax(double income);
 7     }
 8 
 9     // 个人所得税
10     public class PersonalTaxStrategy : ITaxStragety
11     {
12         public double CalculateTax(double income)
13         {
14             return income * 0.12;
15         }
16     }
17 
18     // 企业所得税
19     public class EnterpriseTaxStrategy : ITaxStragety
20     {
21         public double CalculateTax(double income)
22         {
23             return (income - 3500) > 0 ? (income - 3500) * 0.045
24         }
25     }
26 
27     public class InterestOperation
28     {
29         private ITaxStragety m_strategy;
30         public InterestOperation(ITaxStragety strategy)
```

```
31            {
32                this.m_strategy = strategy;
33            }
34
35            public double GetTax(double income)
36            {
37                return m_strategy.CalculateTax(income);
38            }
39        }
40
41        class App
42        {
43            static void Main(string[] args)
44            {
45                // 个人所得税方式
46                InterestOperation operation = new InterestOperation(
47                Console.WriteLine("个人支付的税为：{0}", operation.Get
48
49                // 企业所得税
50                operation = new InterestOperation(new EnterpriseTaxS
51                Console.WriteLine("企业支付的税为：{0}", operation.Get
52
53                Console.Read();
54            }
55        }
56 }
```

三、策略者模式在.NET中应用

在.NET Framework中也不乏策略模式的应用例子。例如，在.NET中，为集合类型ArrayList和List<T>提供的排序功能，其中实现就利用了策略模式，定义了IComparer接口来对比较算法进行封装，实现IComparer接口的类可以是顺序，或逆序地比较两个对象的大小，具体.NET中的实现可以使用反编译工具查看List<T>.Sort(IComparer<T>).aspx)的实现。其中List<T>就是承担着环境角色，而IComparer<T>接口承担着抽象策略角色，具体的策略角色就是实现了IComparer<T>接口的类，List<T>类本身实现了存在实现了该接口的类，我们可以自定义继承与该接口的具体策略类。

四、策略者模式的适用场景

在下面的情况下可以考虑使用策略模式：

- 一个系统需要动态地在几种算法中选择一种的情况下。那么这些算法可以包装到一个个具体的算法类里面，并为这些具体的算法类提供一个统一的接口。
- 如果一个对象有很多的行为，如果不使用合适的模式，这些行为就只好使用多重的if-else语句来实现，此时，可以使用策略模式，把这些行为转移到相应的具体策略类里面，就可以避免使用难以维护的多重条件选择语句，并体现面向

对象涉及的概念。

五、策略者模式的优缺点

策略模式的主要优点有：

- 策略类之间可以自由切换。由于策略类都实现同一个接口，所以使它们之间可以自由切换。
- 易于扩展。增加一个新的策略只需要添加一个具体的策略类即可，基本不需要改变原有的代码。
- 避免使用多重条件选择语句，充分体现面向对象设计思想。

策略模式的主要缺点有：

- 客户端必须知道所有的策略类，并自行决定使用哪一个策略类。这点可以考虑使用IOC容器和依赖注入的方式来解决，关于IOC容器和依赖注入（Dependency Inject）的文章可以参考：IoC 容器和Dependency Injection 模式。
- 策略模式会造成很多的策略类。

六、总结

到这里，策略模式的介绍就结束了，策略模式主要是对方法的封装，把一系列方法封装到一系列的策略类中，从而使不同的策略类可以自由切换和避免在系统使用多重条件选择语句来选择针对不同情况来选择不同的方法。在下一章将会大家介绍责任链模式。

C#设计模式(21)——责任链模式

一、引言

在现实生活中，有很多请求并不是一个人说了就算的，例如面试时的工资，低于1万的薪水可能技术经理就可以决定了，但是1万~1万5的薪水可能技术经理就没这个权利批准，可能就需要请求技术总监的批准，所以在面试的完后，经常会有面试官说，你这个薪水我这边觉得你这技术可以拿这个薪水的，但是还需要技术总监的批准等的话。这个例子也就诠释了本文要介绍的内容。生活中的这个例子真是应用了责任链模式。

二、责任链模式介绍

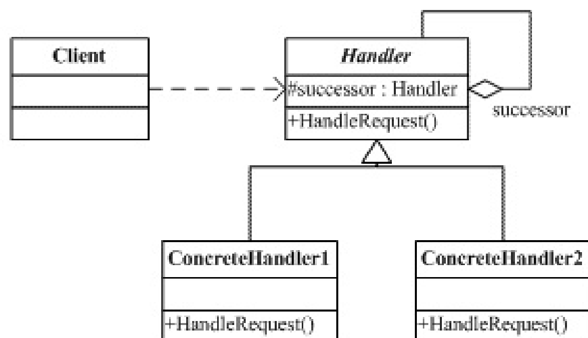
2.1 责任链模式的定义

从生活中的例子可以发现，某个请求可能需要几个人的审批，即使技术经理审批完了，还需要上一级的审批。这样的例子，还有公司中的请假，少于3天的，直属Leader就可以批准，3天到7天之内就需要项目经理批准，多余7天的就需要技术总监的批准了。介绍了这么多生活中责任链模式的例子的，下面具体给出面向对象中责任链模式的定义。

责任链模式指的是——某个请求需要多个对象进行处理，从而避免请求的发送者和接收之间的耦合关系。将这些对象连成一条链子，并沿着这条链子传递该请求，直到有对象处理它为止。

2.2 责任链模式的结构图

从责任链模式的定义可以发现，责任链模式涉及的对象只有处理者角色，但由于有多个处理者，它们具有共同的处理请求的方法，所以这里抽象出一个抽象处理者角色进行代码复用。这样分析下来，责任链模式的结构图也就不言而喻了，具体结构图如下所示。



主要涉及两个角色：

- 抽象处理者角色（Handler）：定义出一个处理请求的接口。这个接口通常由接口或抽象类来实现。
- 具体处理者角色（ConcreteHandler）：具体处理者接受到请求后，可以选择将该请求处理掉，或者将请求传给下一个处理者。因此，每个具体处理者需要保存下一个处理者的引用，以便把请求传递下去。

2.3 责任链模式的实现

有了上面的介绍，下面以公司采购东西为例子来实现责任链模式。公司规定，采购架构总价在1万之内，经理级别的人批准即可，总价大于1万小于2万5的则还需要副总进行批准，总价大于2万5小于10万的需要还需要总经理批准，而大于总价大于10万的则需要组织一个会议进行讨论。对于这样一个需求，最直观的方法就是设计一个方法，参数是采购的总价，然后在这个方法内对价格进行调整判断，然后针对不同的条件交给不同级别的人去处理，这样确实可以解决问题，但这样一来，我们就需要多重if-else语句来进行判断，但当加入一个新的条件范围时，我们又不得不去修改原来设计的方法来再添加一个条件判断，这样的设计显然违背了“开-闭”原则。这时候，可以采用责任链模式来解决这样的问题。具体实现代码如下所示。

```
namespace ChainofResponsibility
{
    // 采购请求
    public class PurchaseRequest
    {
        // 金额
        public double Amount { get; set; }
        // 产品名字
        public string ProductName { get; set; }
        public PurchaseRequest(double amount, string productName)
        {
            Amount = amount;
            ProductName = productName;
        }
    }

    // 审批人,Handler
    public abstract class Approver
    {
        public Approver NextApprover { get; set; }
        public string Name { get; set; }
        public Approver(string name)
        {
            this.Name = name;
        }
        public abstract void ProcessRequest(PurchaseRequest request
    }

    // ConcreteHandler
    public class Manager : Approver
```

```
    {
        public Manager(string name)
            : base(name)
        { }

        public override void ProcessRequest(PurchaseRequest request
        {
            if (request.Amount < 10000.0)
            {
                Console.WriteLine("{0}-{1} approved the request of
            }
            else if (NextApprover != null)
            {
                NextApprover.ProcessRequest(request);
            }
        }
    }

    // ConcreteHandler,副总
    public class VicePresident : Approver
    {
        public VicePresident(string name)
            : base(name)
        {
        }
        public override void ProcessRequest(PurchaseRequest request
        {
            if (request.Amount < 25000.0)
            {
                Console.WriteLine("{0}-{1} approved the request of
            }
            else if (NextApprover != null)
            {
                NextApprover.ProcessRequest(request);
            }
        }
    }

    // ConcreteHandler, 总经理
    public class President :Approver
    {
        public President(string name)
            : base(name)
        { }
        public override void ProcessRequest(PurchaseRequest request
        {
            if (request.Amount < 100000.0)
            {
                Console.WriteLine("{0}-{1} approved the request of
            }
            else
            {
                Console.WriteLine("Request需要组织一个会议讨论");
```

```
                }
            }
        }

        class Program
        {
            static void Main(string[] args)
            {
                PurchaseRequest requestTelphone = new PurchaseRequest(4
                PurchaseRequest requestSoftware = new PurchaseRequest(1
                PurchaseRequest requestComputers = new PurchaseRequest(

                Approver manager = new Manager("LearningHard");
                Approver Vp = new VicePresident("Tony");
                Approver Pre = new President("BossTom");

                // 设置责任链
                manager.NextApprover = Vp;
                Vp.NextApprover = Pre;

                // 处理请求
                manager.ProcessRequest(requestTelphone);
                manager.ProcessRequest(requestSoftware);
                manager.ProcessRequest(requestComputers);
                Console.ReadLine();
            }
        }
    }
```

既然，原来的设计会因为价格条件范围的变化而导致不利于扩展，根据“封装变化”的原则，此时我们想的自然是能不能把价格范围细化到不同的类中呢？因为每个价格范围都决定某个批准者，这里就联想到创建多个批准类，这样每个类中只需要针对他自己这个范围的价格判断。这样也就是责任链的最后实现方式了，具体的运行结果如下图所示。



三、责任链模式的适用场景

在以下场景中可以考虑使用责任链模式：

- 一个系统的审批需要多个对象才能完成处理的情况下，例如请假系统等。
- 代码中存在多个if-else语句的情况下，此时可以考虑使用责任链模式来对代码进行重构。

四、责任链模式的优缺点

责任链模式的优点不言而喻，主要有以下点：

- 降低了请求的发送者和接收者之间的耦合。
- 把多个条件判定分散到各个处理类中，使得代码更加清晰，责任更加明确。

责任链模式也具有一定的缺点，如：

- 在找到正确的处理对象之前，所有的条件判定都要执行一遍，当责任链过长时，可能会引起性能的问题
- 可能导致某个请求不被处理。

五、总结

责任链降低了请求端和接收端之间的耦合，使多个对象都有机会处理某个请求。如考试中作弊传纸条，泡妞传情书一般。在下一章将继续分享访问者模式。

C#设计模式(22)——访问者模式（Vistor Pattern）

一、引言

在上一篇博文中分享了责任链模式，责任链模式主要应用在系统中的某些功能需要多个对象参与才能完成的场景。在这篇博文中，我将为大家分享我对访问者模式的理解。

二、访问者模式介绍

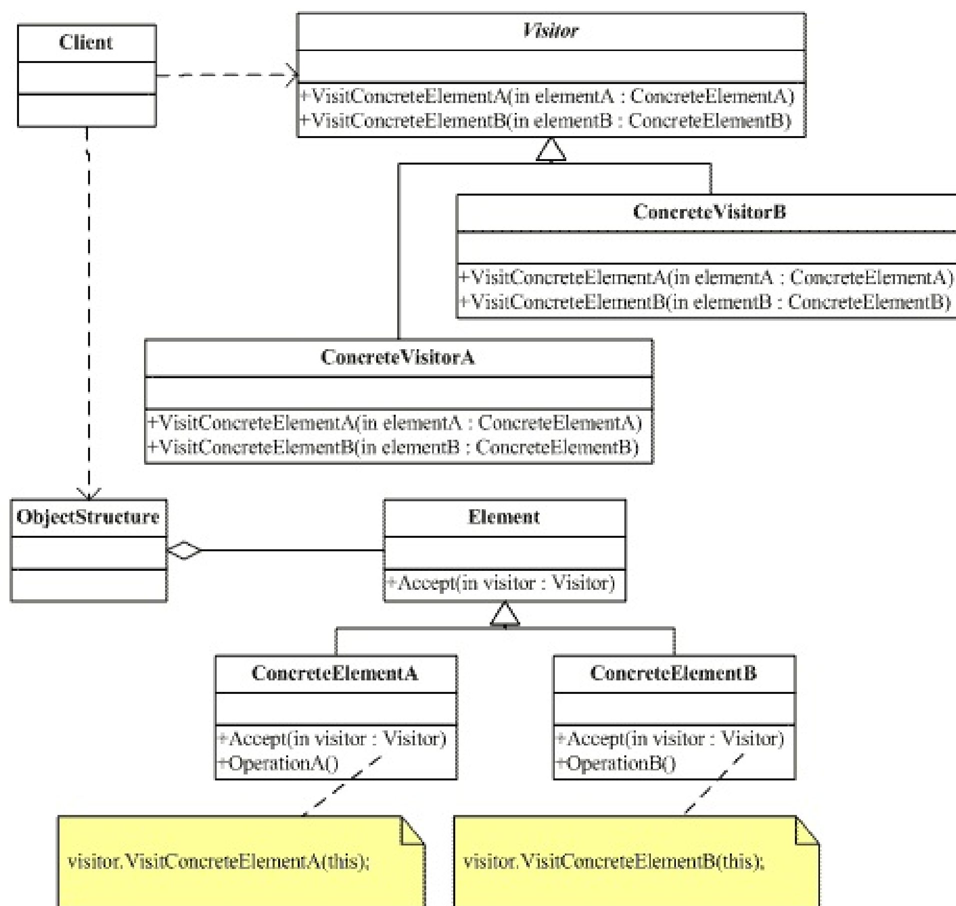
2.1 访问者模式的定义

访问者模式是封装一些施加于某种数据结构之上的操作。一旦这些操作需要修改的话，接受这个操作的数据结构则可以保存不变。访问者模式适用于数据结构相对稳定的系统， 它把数据结构和作用于数据结构之上的操作之间的耦合度降低，使得操作集合可以相对自由地改变。

数据结构的每一个节点都可以接受一个访问者的调用，此节点向访问者对象传入节点对象，而访问者对象则反过来执行节点对象的操作。这样的过程叫做“双重分派”。节点调用访问者，将它自己传入，访问者则将某算法针对此节点执行。

2.2 访问者模式的结构图

从上面描述可知，访问者模式是用来封装某种数据结构中的方法。具体封装过程是：每个元素接受一个访问者的调用，每个元素的Accept方法接受访问者对象作为参数传入，访问者对象则反过来调用元素对象的操作。具体的访问者模式结构图如下所示。



这里需要明确一点：访问者模式中具体访问者的数目和具体节点的数目没有任何关系。从访问者的结构图可以看出，访问者模式涉及以下几类角色。

- 抽象访问者角色（Vistor）:声明一个活多个访问操作，使得所有具体访问者必须实现的接口。
- 具体访问者角色（ConcreteVistor）：实现抽象访问者角色中所有声明的接口。
- 抽象节点角色（Element）：声明一个接受操作，接受一个访问者对象作为参数。
- 具体节点角色（ConcreteElement）：实现抽象元素所规定的接受操作。
- 结构对象角色（ObjectStructure）：节点的容器，可以包含多个不同类或接口的容器。

2.3 访问者模式的实现

在讲诉访问者模式的实现时，我想先不用访问者模式的方式来实现某个场景。具体场景是——现在我想遍历每个元素对象，然后调用每个元素对象的Print方法来打印该元素对象的信息。如果此时不采用访问者模式的话，实现这个场景再简单不过了，具体实现代码如下所示：

```
1 namespace DonotUsevistorPattern
2 {
3     // 抽象元素角色
```

```
4       public abstract class Element
5       {
6           public abstract void Print();
7       }
8
9       // 具体元素A
10      public class ElementA : Element
11      {
12          public override void Print()
13          {
14              Console.WriteLine("我是元素A");
15          }
16      }
17
18      // 具体元素B
19      public class ElementB : Element
20      {
21          public override void Print()
22          {
23              Console.WriteLine("我是元素B");
24          }
25      }
26
27      // 对象结构
28      public class ObjectStructure
29      {
30          private ArrayList elements = new ArrayList();
31
32          public ArrayList Elements
33          {
34              get { return elements; }
35          }
36
37          public ObjectStructure()
38          {
39              Random ran = new Random();
40              for (int i = 0; i < 6; i++)
41              {
42                  int ranNum = ran.Next(10);
43                  if (ranNum > 5)
44                  {
45                      elements.Add(new ElementA());
46                  }
47                  else
48                  {
49                      elements.Add(new ElementB());
50                  }
51              }
52          }
53      }
54
55      class Program
56      {
```

```
57          static void Main(string[] args)
58          {
59              ObjectStructure objectStructure = new ObjectStructur
60              // 遍历对象结构中的对象集合，访问每个元素的Print方法打印元素
61              foreach (Element e in objectStructure.Elements)
62              {
63                  e.Print();
64              }
65
66              Console.Read();
67          }
68      }
69  }
```

上面代码很准确了解决了我们刚才提出的场景，但是需求在时刻变化的，如果此时，我除了想打印元素的信息外，还想打印出元素被访问的时间，此时我们就不得不去修改每个元素的Print方法，再加入相对应的输入访问时间的输出信息。这样的设计显然不符合“开-闭”原则，即某个方法操作的改变，会使得必须去更改每个元素类。既然，这里变化的点是操作的改变，而每个元素的数据结构是不变的。所以此时就思考——能不能把操作于元素的操作和元素本身的数据结构分开呢？解开这两者的耦合度，这样如果是操作发现变化时，就不需要去更改元素本身了，但是如果是元素数据结构发现变化，例如，添加了某个字段，这样就不得不去修改元素类了。此时，我们可以使用访问者模式来解决这个问题，即把作用于具体元素的操作由访问者对象来调用。具体的实现代码如下所示：

```
 1 namespace VistorPattern
 2 {
 3     // 抽象元素角色
 4     public abstract class Element
 5     {
 6         public abstract void Accept(IVistor vistor);
 7         public abstract void Print();
 8     }
 9
10     // 具体元素A
11     public class ElementA :Element
12     {
13         public override void Accept(IVistor vistor)
14         {
15             // 调用访问者visit方法
16             vistor.Visit(this);
17         }
18         public override void Print()
19         {
20             Console.WriteLine("我是元素A");
21         }
22     }
23
24     // 具体元素B
25     public class ElementB :Element
```

```
26      {
27          public override void Accept(IVistor vistor)
28          {
29              vistor.Visit(this);
30          }
31          public override void Print()
32          {
33              Console.WriteLine("我是元素B");
34          }
35      }
36
37      // 抽象访问者
38      public interface IVistor
39      {
40          void Visit(ElementA a);
41          void Visit(ElementB b);
42      }
43
44      // 具体访问者
45      public class ConcreteVistor :IVistor
46      {
47          // visit方法而是再去调用元素的Accept方法
48          public void Visit(ElementA a)
49          {
50              a.Print();
51          }
52          public void Visit(ElementB b)
53          {
54              b.Print();
55          }
56      }
57
58      // 对象结构
59      public class ObjectStructure
60      {
61          private ArrayList elements = new ArrayList();
62
63          public ArrayList Elements
64          {
65              get { return elements; }
66          }
67
68          public ObjectStructure()
69          {
70              Random ran = new Random();
71              for (int i = 0; i < 6; i++)
72              {
73                  int ranNum = ran.Next(10);
74                  if (ranNum > 5)
75                  {
76                      elements.Add(new ElementA());
77                  }
78                  else
```

```
 79                 {
 80                     elements.Add(new ElementB());
 81                 }
 82             }
 83         }
 84     }
 85 
 86     class Program
 87     {
 88         static void Main(string[] args)
 89         {
 90             ObjectStructure objectStructure = new ObjectStructu
 91             foreach (Element e in objectStructure.Elements)
 92             {
 93                 // 每个元素接受访问者访问
 94                 e.Accept(new ConcreteVistor());
 95             }
 96 
 97             Console.Read();
 98         }
 99     }
100 }
```

从上面代码可知，使用访问者模式实现上面场景后，元素Print方法的访问封装到了访问者对象中了（我觉得可以把Print方法封装到具体访问者对象中。），此时客户端与元素的Print方法就隔离开了。此时，如果需要添加打印访问时间的需求时，此时只需要再添加一个具体的访问者类即可。此时就不需要去修改元素中的Print()方法了。

三、访问者模式的应用场景

每个设计模式都有其应当使用的情况，那让我们看看访问者模式具体应用场景。如果遇到以下场景，此时我们可以考虑使用访问者模式。

- 如果系统有比较稳定的数据结构，而又有易于变化的算法时，此时可以考虑使用访问者模式。因为访问者模式使得算法操作的添加比较容易。
- 如果一组类中，存在着相似的操作，为了避免出现大量重复的代码，可以考虑把重复的操作封装到访问者中。（当然也可以考虑使用抽象类了）
- 如果一个对象存在着一些与本身对象不相干，或关系比较弱的操作时，为了避免操作污染这个对象，则可以考虑把这些操作封装到访问者对象中。

四、访问者模式的优缺点

访问者模式具有以下优点：

- 访问者模式使得添加新的操作变得容易。如果一些操作依赖于一个复杂的结构对象的话，那么一般而言，添加新的操作会变得很复杂。而使用访问者模式，

增加新的操作就意味着添加一个新的访问者类。因此，使得添加新的操作变得容易。
- 访问者模式使得有关的行为操作集中到一个访问者对象中，而不是分散到一个个的元素类中。这点类似与"中介者模式"。
- 访问者模式可以访问属于不同的等级结构的成员对象，而迭代只能访问属于同一个等级结构的成员对象。

访问者模式也有如下的缺点：

- 增加新的元素类变得困难。每增加一个新的元素意味着要在抽象访问者角色中增加一个新的抽象操作，并在每一个具体访问者类中添加相应的具体操作。

五、总结

访问者模式是用来封装一些施加于某种数据结构之上的操作。它使得可以在不改变元素本身的前提下增加作用于这些元素的新操作，访问者模式的目的是把操作从数据结构中分离出来。

C#设计模式(23)——备忘录模式（Memento Pattern）

一、引言

在上一篇博文分享了访问者模式，访问者模式的实现是把作用于某种数据结构上的操作封装到访问者中，使得操作和数据结构隔离。而今天要介绍的备忘者模式与命令模式有点相似，不同的是，命令模式保存的是发起人的具体命令（命令对应的是行为），而备忘录模式保存的是发起人的状态（而状态对应的数据结构，如属性）。下面具体来看看备忘录模式。

二、备忘录模式介绍

2.1 备忘录模式的定义

从字面意思就可以明白，备忘录模式就是对某个类的状态进行保存下来，等到需要恢复的时候，可以从备忘录中进行恢复。生活中这样的例子经常看到，如备忘电话通讯录，备份操作操作系统，备份数据库等。

备忘录模式的具体定义是：在不破坏封装的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，这样以后就可以把该对象恢复到原先的状态。

2.2 备忘录模式的结构图

介绍完备忘录模式的定义之后，下面具体看看备忘录模式的结构图：



备忘录模式中主要有三类角色：

- 发起人角色：记录当前时刻的内部状态，负责创建和恢复备忘录数据。
- 备忘录角色：负责存储发起人对象的内部状态，在进行恢复时提供给发起人需要的状态。
- 管理者角色：负责保存备忘录对象。

2.3 备忘录模式的实现

下面以备份手机通讯录为例子来实现了备忘录模式，具体的实现代码如下所示：

```
// 联系人
    public class ContactPerson
    {
        public string Name { get; set; }
        public string MobileNum { get; set; }
    }

    // 发起人
    public class MobileOwner
    {
        // 发起人需要保存的内部状态
        public List<ContactPerson> ContactPersons { get; set; }

        public MobileOwner(List<ContactPerson> persons)
        {
            ContactPersons = persons;
        }

        // 创建备忘录，将当期要保存的联系人列表导入到备忘录中
        public ContactMemento CreateMemento()
        {
            // 这里也应该传递深拷贝，new List方式传递的是浅拷贝，
            // 因为ContactPerson类中都是string类型,所以这里new list方式
            // 如果ContactPerson包括非string的引用类型就会有问题，所以这里
            return new ContactMemento(new List<ContactPerson>(this.
        }

        // 将备忘录中的数据备份导入到联系人列表中
        public void RestoreMemento(ContactMemento memento)
        {
            // 下面这种方式是错误的，因为这样传递的是引用，
            // 则删除一次可以恢复，但恢复之后再删除的话就恢复不了.
            // 所以应该传递contactPersonBack的深拷贝，深拷贝可以使用序列化
            this.ContactPersons = memento.contactPersonBack;
        }

        public void Show()
        {
            Console.WriteLine("联系人列表中有{0}个人，他们是:", Contact
            foreach (ContactPerson p in ContactPersons)
            {
                Console.WriteLine("姓名: {0} 号码为: {1}", p.Name, p
            }
        }
    }

    // 备忘录
    public class ContactMemento
    {
        // 保存发起人的内部状态
```

```
        public List<ContactPerson> contactPersonBack;

        public ContactMemento(List<ContactPerson> persons)
        {
            contactPersonBack = persons;
        }
    }

    // 管理角色
    public class Caretaker
    {
        public ContactMemento ContactM { get; set; }
    }

    class Program
    {
        static void Main(string[] args)
        {
            List<ContactPerson> persons = new List<ContactPerson>()
            {
                new ContactPerson() { Name= "Learning Hard", Mobile
                new ContactPerson() { Name = "Tony", MobileNum = "2
                new ContactPerson() { Name = "Jock", MobileNum = "2
            };
            MobileOwner mobileOwner = new MobileOwner(persons);
            mobileOwner.Show();

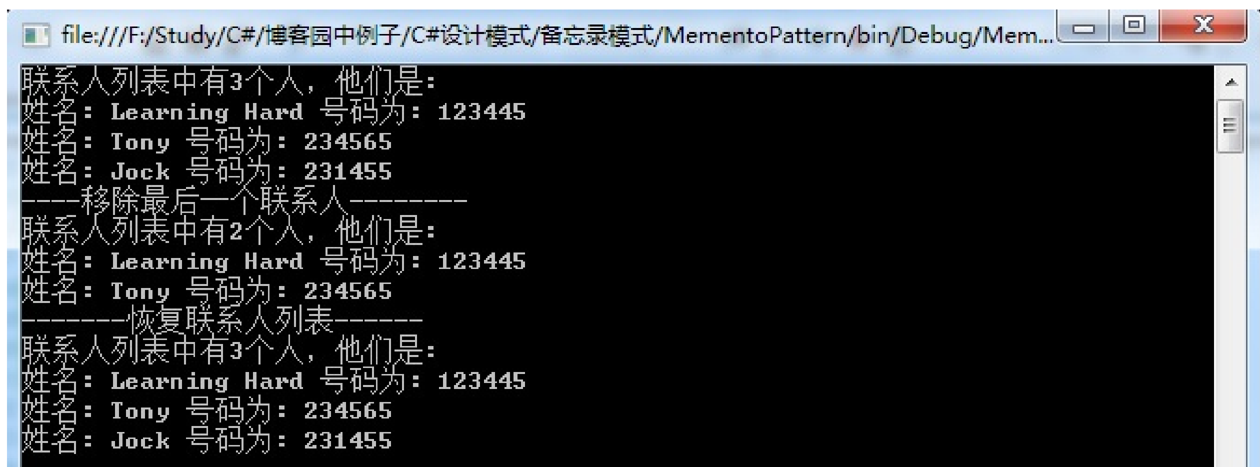
            // 创建备忘录并保存备忘录对象
            Caretaker caretaker = new Caretaker();
            caretaker.ContactM = mobileOwner.CreateMemento();

            // 更改发起人联系人列表
            Console.WriteLine("----移除最后一个联系人--------");
            mobileOwner.ContactPersons.RemoveAt(2);
            mobileOwner.Show();

            // 恢复到原始状态
            Console.WriteLine("-------恢复联系人列表------");
            mobileOwner.RestoreMemento(caretaker.ContactM);
            mobileOwner.Show();

            Console.Read();
        }
    }
```

具体的运行结果如下图所示：



从上图可以看出，刚开始通讯录中有3个联系人，然后移除以后一个后变成2个联系人了，最后恢复原来的联系人列表后，联系人列表中又恢复为3个联系人了。

上面代码只是保存了一个还原点，即备忘录中只保存了3个联系人的数据，但是，如果想备份多个还原点怎么办呢？即恢复到3个人后，又想恢复到前面2个人的状态，这时候可能你会想，这样没必要啊，到时候在删除不就好了。但是如果在实际应用中，可能我们发了很多时间去创建通讯录中只有2个联系人的状态，恢复到3个人的状态后，发现这个状态时错误的，还是原来2个人的状态是正确的，难道我们又去花之前的那么多时间去重复操作吗？这显然不合理，如果就思考，能不能保存多个还原点呢？保存多个还原点其实很简单，只需要保存多个备忘录对象就可以了。具体实现代码如下所示：

```
namespace MultipleMementoPattern
{
    // 联系人
    public class ContactPerson
    {
        public string Name { get; set; }
        public string MobileNum { get; set; }
    }

    // 发起人
    public class MobileOwner
    {
        public List<ContactPerson> ContactPersons { get; set; }
        public MobileOwner(List<ContactPerson> persons)
        {
            ContactPersons = persons;
        }

        // 创建备忘录，将当期要保存的联系人列表导入到备忘录中
        public ContactMemento CreateMemento()
        {
             // 这里也应该传递深拷贝，new List方式传递的是浅拷贝，
            // 因为ContactPerson类中都是string类型,所以这里new list方式
            // 如果ContactPerson包括非string的引用类型就会有问题，所以这里
            return new ContactMemento(new List<ContactPerson>(this
        }
```

```
        // 将备忘录中的数据备份导入到联系人列表中
        public void RestoreMemento(ContactMemento memento)
        {
            if (memento != null)
            {
                // 下面这种方式是错误的，因为这样传递的是引用，
                // 则删除一次可以恢复，但恢复之后再删除的话就恢复不了.
                // 所以应该传递contactPersonBack的深拷贝，深拷贝可以使用Б
                this.ContactPersons = memento.ContactPersonBack;
            }
        }
        public void Show()
        {
            Console.WriteLine("联系人列表中有{0}个人，他们是:", Contact
            foreach (ContactPerson p in ContactPersons)
            {
                Console.WriteLine("姓名: {0} 号码为: {1}", p.Name, p
            }
        }
    }

    // 备忘录
    public class ContactMemento
    {
        public List<ContactPerson> ContactPersonBack {get;set;}
        public ContactMemento(List<ContactPerson> persons)
        {
            ContactPersonBack = persons;
        }
    }

    // 管理角色
    public class Caretaker
    {
        // 使用多个备忘录来存储多个备份点
        public Dictionary<string, ContactMemento> ContactMementoDic
        public Caretaker()
        {
            ContactMementoDic = new Dictionary<string, ContactMemen
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            List<ContactPerson> persons = new List<ContactPerson>()
            {
                new ContactPerson() { Name= "Learning Hard", Mobile
                new ContactPerson() { Name = "Tony", MobileNum = "2
                new ContactPerson() { Name = "Jock", MobileNum = "2
            };
```

```
            MobileOwner mobileOwner = new MobileOwner(persons);
            mobileOwner.Show();

            // 创建备忘录并保存备忘录对象
            Caretaker caretaker = new Caretaker();
            caretaker.ContactMementoDic.Add(DateTime.Now.ToString()

            // 更改发起人联系人列表
            Console.WriteLine("----移除最后一个联系人--------");
            mobileOwner.ContactPersons.RemoveAt(2);
            mobileOwner.Show();

            // 创建第二个备份
            Thread.Sleep(1000);
            caretaker.ContactMementoDic.Add(DateTime.Now.ToString()

            // 恢复到原始状态
            Console.WriteLine("-------恢复联系人列表,请从以下列表选择恢复
            var keyCollection = caretaker.ContactMementoDic.Keys;
            foreach (string k in keyCollection)
            {
                Console.WriteLine("Key = {0}", k);
            }
            while (true)
            {
                Console.Write("请输入数字,按窗口的关闭键退出:");

                int index = -1;
                try
                {
                    index = Int32.Parse(Console.ReadLine());
                }
                catch
                {
                    Console.WriteLine("输入的格式错误");
                    continue;
                }

                ContactMemento contactMentor = null;
                if (index < keyCollection.Count && caretaker.Contac
                {
                    mobileOwner.RestoreMemento(contactMentor);
                    mobileOwner.Show();
                }
                else
                {
                    Console.WriteLine("输入的索引大于集合长度！");
                }
            }
        }
    }
}
```

这样就保存了多个状态，客户端可以选择恢复的状态点，具体运行结果如下所示：

```
file:///F:/Study/C#/博客园中例子/C#设计模式/备忘录模式/MultipleMementoPattern/bin/Deb...
联系人列表中有3个人，他们是:
姓名: Learning Hard 号码为: 123445
姓名: Tony 号码为: 234565
姓名: Jock 号码为: 231455
----移除最后一个联系人--------
联系人列表中有2个人，他们是:
姓名: Learning Hard 号码为: 123445
姓名: Tony 号码为: 234565
-------恢复联系人列表,请从以下列表选择恢复的日期------
Key = 2014/9/20 18:06:09
Key = 2014/9/20 18:06:10
请输入数字,按窗口的关闭键退出:0
联系人列表中有3个人，他们是:
姓名: Learning Hard 号码为: 123445
姓名: Tony 号码为: 234565
姓名: Jock 号码为: 231455
请输入数字,按窗口的关闭键退出:1
联系人列表中有2个人，他们是:
姓名: Learning Hard 号码为: 123445
姓名: Tony 号码为: 234565
请输入数字,按窗口的关闭键退出:
```

三、备忘录模式的适用场景

在以下情况下可以考虑使用备忘录模式：

- 如果系统需要提供回滚操作时，使用备忘录模式非常合适。例如文本编辑器的Ctrl+Z撤销操作的实现，数据库中事务操作。

四、备忘录模式的优缺点

备忘录模式具有以下优点：

- 如果某个操作错误地破坏了数据的完整性，此时可以使用备忘录模式将数据恢复成原来正确的数据。
- 备份的状态数据保存在发起人角色之外，这样发起人就不需要对各个备份的状态进行管理。而是由备忘录角色进行管理，而备忘录角色又是由管理者角色管理，符合单一职责原则。

当然，备忘录模式也存在一定的缺点：

- 在实际的系统中，可能需要维护多个备份，需要额外的资源，这样对资源的消耗比较严重。

五、总结

备忘录模式主要思想是——利用备忘录对象来对保存发起人的内部状态，当发起人需要恢复原来状态时，再从备忘录对象中进行获取，在实际开发过程也应用到这点，例如数据库中的事务处理。

C#设计模式总结

一、引言

经过这段时间对设计模式的学习，自己的感触还是很多的，因为我现在在写代码的时候，经常会想想这里能不能用什么设计模式来进行重构。所以，学完设计模式之后，感觉它会慢慢地影响到你写代码的思维方式。这里对设计模式做一个总结，一来可以对所有设计模式进行一个梳理，二来可以做一个索引来帮助大家收藏。

PS: 其实，很早之前我就看过所有的设计模式了，但是并没有写博客，但是不久就很快忘记了，也没有起到什么作用，这次以博客的形式总结出来，发现效果还是很明显的，因为通过这种总结的方式，我对它理解更深刻了，也记住的更牢靠了，也影响了自己平时实现功能的思维。所以，我鼓励大家可以通过做笔记的方式来把自己学到的东西进行梳理，这样相信可以理解更深，更好，我也会一直写下来，之后打算写WCF一系列文章。

其实WCF内容很早也看过了，并且博客园也有很多前辈写的很好，但是，我觉得我还是需要自己总结，因为只有这样，知识才是自己的，别人写的多好，你看了之后，其实还是别人了，所以鼓励大家几点（对于这几点，也是对自己的一个提醒）：

1. 要动手实战别人博客中的例子；
2. 实现之后进行总结，可以写博客也可以自己记录云笔记等；
3. 想想能不能进行扩展，进行举一反三。

系列导航：

C#设计模式(1)——单例模式

C#设计模式(2)——简单工厂模式

C#设计模式(3)——工厂方法模式

C#设计模式(4)——抽象工厂模式

C#设计模式(5)——建造者模式（Builder Pattern）

C#设计模式(6)——原型模式（Prototype Pattern）

C#设计模式(7)——适配器模式（Adapter Pattern）

C#设计模式(8)——桥接模式（Bridge Pattern）

C#设计模式(9)——装饰者模式（Decorator Pattern）

C#设计模式(10)——组合模式（Composite Pattern）

C#设计模式(11)——外观模式（Facade Pattern）

二、 设计原则

使用设计模式的根本原因是适应变化，提高代码复用率，使软件更具有可维护性和可扩展性。并且，在进行设计的时候，也需要遵循以下几个原则：单一职责原则、开放封闭原则、里氏代替原则、依赖倒置原则、接口隔离原则、合成复用原则和迪米特法则。下面就分别介绍了每种设计原则。

2.1 单一职责原则

就一个类而言，应该只有一个引起它变化的原因。如果一个类承担的职责过多，就等于把这些职责耦合在一起，一个职责的变化可能会影响到其他的职责，另外，把多个职责耦合在一起，也会影响复用性。

2.2 开闭原则(Open-Closed Principle)

开闭原则即OCP（Open-Closed Principle缩写）原则，该原则强调的是：一个软件实体（指的类、函数、模块等）应该对扩展开放，对修改关闭。即每次发生变化时，要通过添加新的代码来增强现有类型的行为，而不是修改原有的代码。

符合开闭原则的最好方式是提供一个固有的接口，然后让所有可能发生变化的类实现该接口，让固定的接口与相关对象进行交互。

2.3 里氏代替原则(Liskov Substitution Principle)

Liskov Substitution Principle,LSP（里氏代替原则）指的是子类必须替换掉它们的父类型。也就是说，在软件开发过程中，子类替换父类后，程序的行为是一样的。只有当子类替换掉父类后，此时软件的功能不受影响时，父类才能真正地被复用，而子类也可以在父类的基础上添加新的行为。为了就来看看违反了LSP原则的例子，具体代码如下所示：

```
public class Rectangle
   {
       public virtual long Width { get; set; }
       public virtual long Height { get; set; }
   }
   // 正方形
   public class Square : Rectangle
   {
       public override long Height
       {
           get
           {
               return base.Height;
           }
           set
           {
               base.Height = value;
               base.Width = value;
           }
       }

       public override long Width
       {
           get
           {
               return base.Width;
           }
           set
           {
               base.Width = value;
               base.Height = value;
           }
       }
   }
class Test
   {
       public void Resize(Rectangle r)
       {
           while (r.Height >= r.Width)
           {
               r.Width += 1;
           }
       }
       var r = new Square() { Width = 10, Height = 10 };
        new Test().Resize(r);
    }
```

上面的设计，正如上面注释的一样，在执行SmartTest的resize方法时，如果传入的是长方形对象，当高度大于宽度时，会自动增加宽度直到超出高度。但是如果传入的是正方形对象，则会陷入死循环。此时根本原因是，矩形不能作为正方形的父

类，既然出现了问题，可以进行重构，使它们俩都继承于四边形类。重构后的代码如下所示：

```
// 四边形
    public abstract class Quadrangle
    {
        public virtual long Width { get; set; }
        public virtual long Height { get; set; }
    }
    // 矩形
    public class Rectangle : Quadrangle
    {
        public override long Height { get; set; }

        public override long Width { get; set; }

    }
    // 正方形
    public class Square : Quadrangle
    {
        public long _side;

        public Square(long side)
        {
            _side = side;
        }
    }
class Test
    {
        public void Resize(Quadrangle r)
        {
            while (r.Height >= r.Width)
            {
                r.Width += 1;
            }
        }

        static void Main(string[] args)
        {
            var s = new Square(10);

            new Test().Resize(s);
        }
    }
```

2.4 依赖倒置原则

依赖倒置（Dependence Inversion Principle, DIP）原则指的是抽象不应该依赖于细节，细节应该依赖于抽象，也就是提出的 “面向接口编程，而不是面向实现编程”。这样可以降低客户与具体实现的耦合。

2.5 接口隔离原则

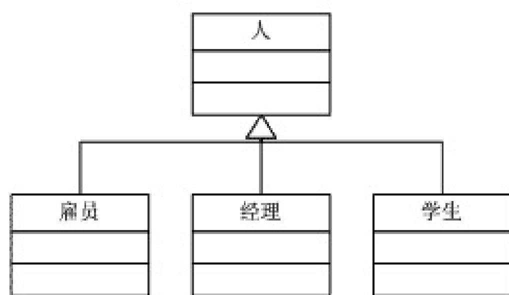
接口隔离原则（Interface Segregation Principle, ISP）指的是使用多个专门的接口比使用单一的总接口要好。也就是说不要让一个单一的接口承担过多的职责，而应把每个职责分离到多个专门的接口中，进行接口分离。过于臃肿的接口是对接口的一种污染。

2.6 合成复用原则

合成复用原则（Composite Reuse Principle, CRP）就是在一个新的对象里面使用一些已有的对象，使之成为新对象的一部分。新对象通过向这些对象的委派达到复用已用功能的目的。简单地说，就是要尽量使用合成/聚合，尽量不要使用继承。

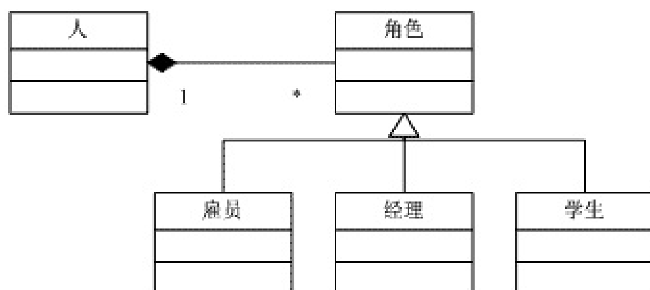
要使用好合成复用原则，首先需要区分"Has—A"和“Is—A”的关系。

“Is—A”是指一个类是另一个类的“一种”，是属于的关系，而“Has—A”则不同，它表示某一个角色具有某一项责任。导致错误的使用继承而不是聚合的常见的原因是错误地把“Has—A”当成“Is—A”.例如：



实际上，雇员、经历、学生描述的是一种角色，比如一个人是“经理”必然是“雇员”。在上面的设计中，一个人无法同时拥有多个角色，是“雇员”就不能再是“学生”了，这显然不合理，因为现在很多在职研究生，即使雇员也是学生。

上面的设计的错误源于把“角色”的等级结构与“人”的等级结构混淆起来了，误把“Has—A”当作"Is—A"。具体的解决方法就是抽象出一个角色类：



2.7 迪米特法则

迪米特法则（Law of Demeter，LoD）又叫最少知识原则（Least Knowledge Principle，LKP），指的是一个对象应当对其他对象有尽可能少的了解。也就是说，一个模块或对象应尽量少的与其他实体之间发生相互作用，使得系统功能模块相对独立，这样当一个模块修改时，影响的模块就会越少，扩展起来更加容易。

关于迪米特法则其他的一些表述有：只与你直接的朋友们通信；不要跟“陌生人”说话。

外观模式（Facade Pattern)和中介者模式（Mediator Pattern）就使用了迪米特法则。

三、创建型模式

创建型模式就是用来创建对象的模式，抽象了实例化的过程。所有的创建型模式都有两个共同点。第一，它们都将系统使用哪些具体类的信息封装起来；第二，它们隐藏了这些类的实例是如何被创建和组织的。创建型模式包括单例模式、工厂方法模式、抽象工厂模式、建造者模式和原型模式。

- 单例模式：解决的是实例化对象的个数的问题，比如抽象工厂中的工厂、对象池等，除了Singleton之外，其他创建型模式解决的都是 new 所带来的耦合关系。
- 抽象工厂：创建一系列相互依赖对象，并能在运行时改变系列。
- 工厂方法：创建单个对象，在Abstract Factory有使用到。
- 原型模式：通过拷贝原型来创建新的对象。

工厂方法，抽象工厂, 建造者都需要一个额外的工厂类来负责实例化“一个对象”，而Prototype则是通过原型（一个特殊的工厂类）来克隆“易变对象”。

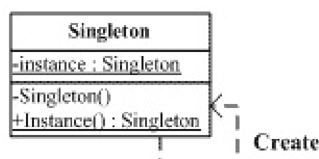
下面详细介绍下它们。

3.1 单例模式

单例模式指的是确保某一个类只有一个实例，并提供一个全局访问点。解决的是实体对象个数的问题，而其他的建造者模式都是解决new所带来的耦合关系问题。其实现要点有：

- 类只有一个实例。问：如何保证呢？答：通过私有构造函数来保证类外部不能对类进行实例化
- 提供一个全局的访问点。问：如何实现呢？答：创建一个返回该类对象的静态方法

单例模式的结构图如下所示：

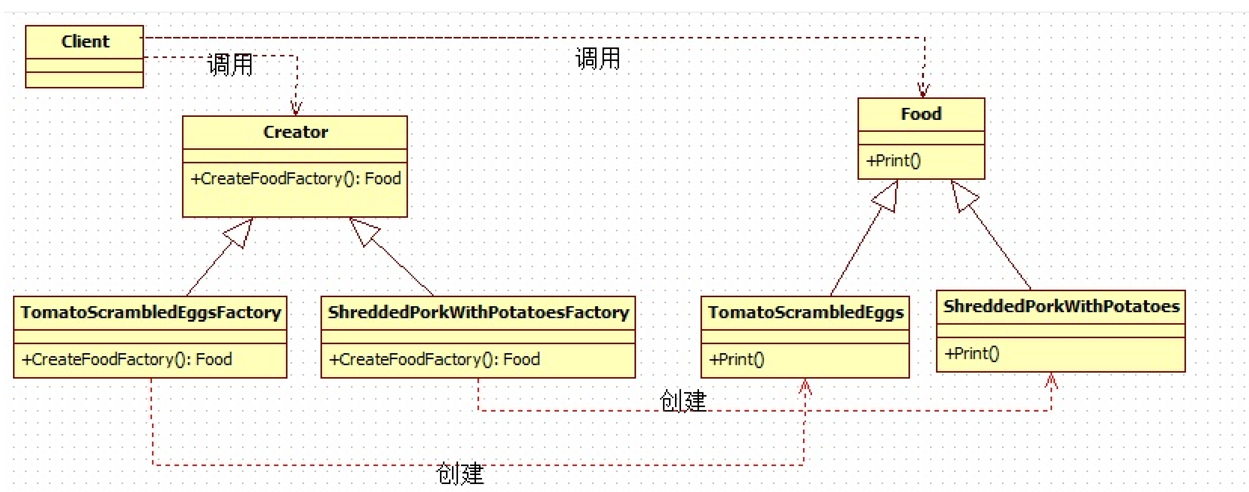


3.2 工厂方法模式

工厂方法模式指的是定义一个创建对象的工厂接口，由其子类决定要实例化的类，将实际创建工作推迟到子类中。它强调的是"单个对象"的变化。其实现要点有：

- 定义一个工厂接口。问：如何实现呢？答：声明一个工厂抽象类
- 由其具体子类创建对象。问：如何去实现呢？答：创建派生于工厂抽象类，即由具体工厂去创建具体产品，既然要创建产品，自然需要产品抽象类和具体产品类了。

其具体的UML结构图如下所示：



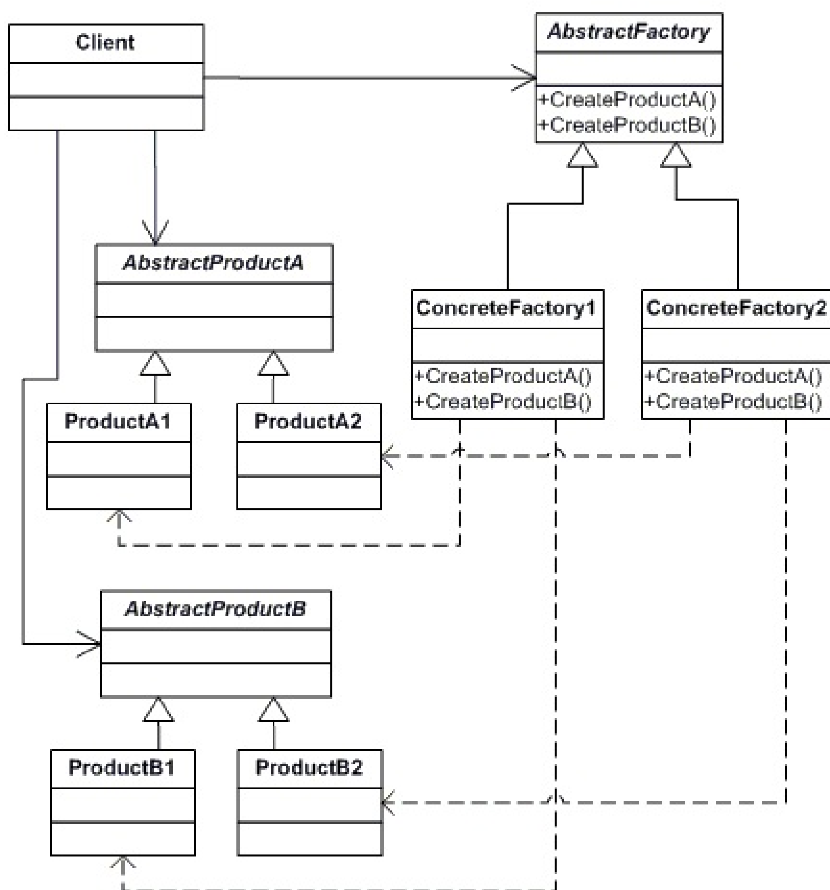
在工厂方法模式中，工厂类与具体产品类具有平行的等级结构，它们之间是一一对应关系。

3.3 抽象工厂模式

抽象工厂模式指的是提供一个创建一系列相关或相互依赖对象的接口，使得客户端可以在不必指定产品的具体类型的情况下，创建多个产品族中的产品对象，强调的是"系列对象"的变化。其实现要点有：

- 提供一系列对象的接口。问：如何去实现呢？答：提供多个产品的抽象接口
- 创建多个产品族中的多个产品对象。问：如何做到呢？答：每个具体工厂创建一个产品族中的多个产品对象，多个具体工厂就可以创建多个产品族中的多个对象了。

具体的UML结构图如下所示：

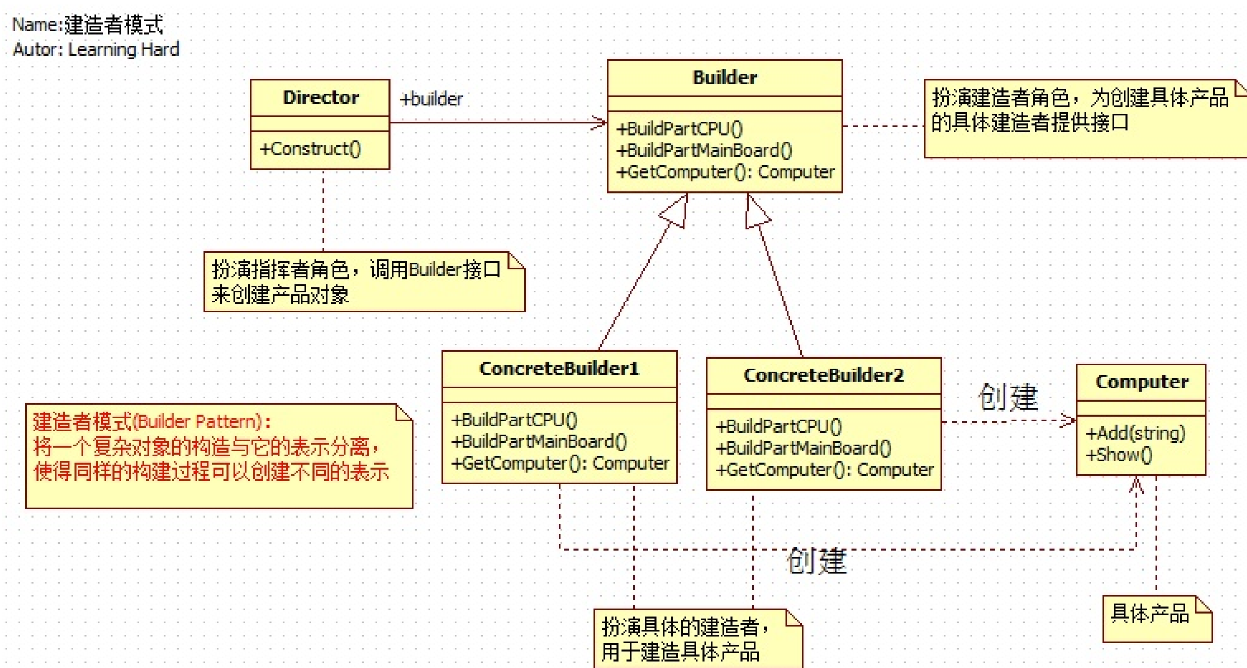


3.4 建造者模式

建造者模式指的是将一个产品的内部表示与产品的构造过程分割开来，从而可以使一个建造过程生成具体不同的内部表示的产品对象。强调的是产品的构造过程。其实现要点有：

- 将产品的内部表示与产品的构造过程分割开来。问：如何把它们分割开呢？答：不要把产品的构造过程放在产品类中，而是由建造者类来负责构造过程，产品的内部表示放在产品类中，这样不就分割开了嘛。

具体的UML结构图如下所示：

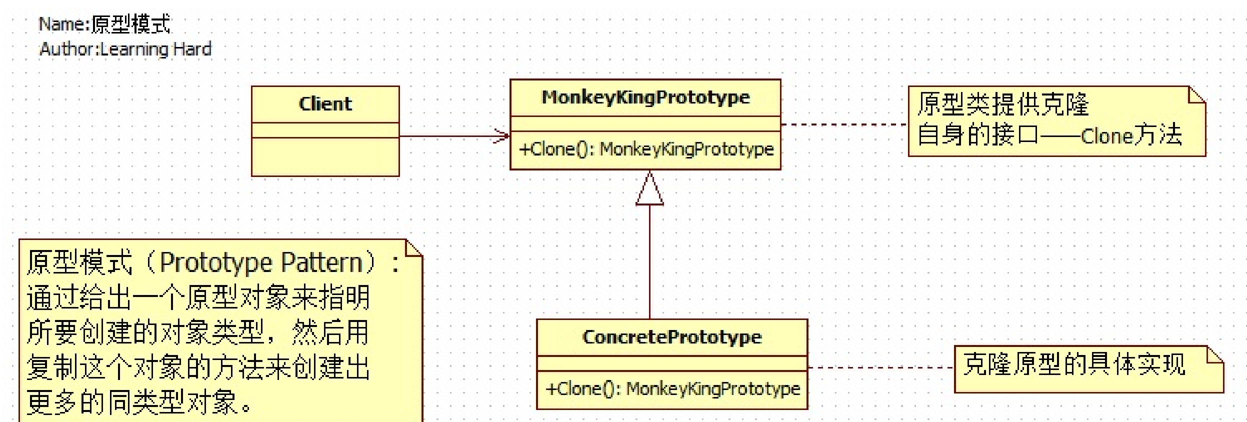


3.5 原型工厂模式

原型模式指的是通过给出一个原型对象来指明所要创建的对象类型，然后用复制的方法来创建出更多的同类型对象。其实现要点有：

- 给出一个原型对象。问：如何办到呢？答：很简单嘛，直接给出一个原型类就好了。
- 通过复制的方法来创建同类型对象。问：又是如何实现呢？答：.NET可以直接调用MemberwiseClone方法来实现浅拷贝

具体的UML结构图如下所示：



四、结构型模式

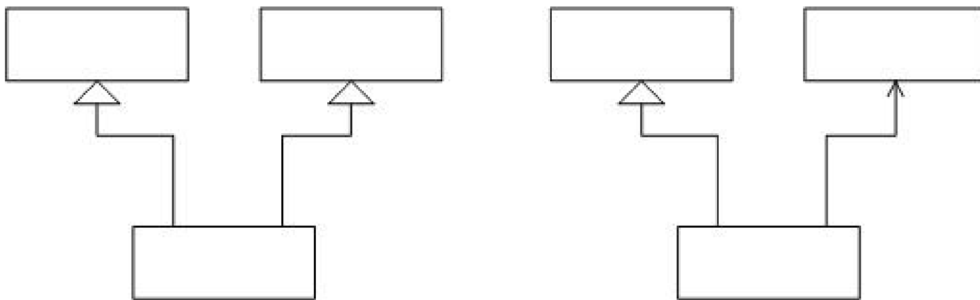
结构型模式，顾名思义讨论的是类和对象的结构，主要用来处理类或对象的组合。它包括两种类型，一是类结构型模式，指的是采用继承机制来组合接口或实现；二是对象结构型模式，指的是通过组合对象的方式来实现新的功能。它包括适配器模

式、桥接模式、装饰者模式、组合模式、外观模式、享元模式和代理模式。

- 适配器模式注重转换接口，将不吻合的接口适配对接
- 桥接模式注重分离接口与其实现，支持多维度变化
- 组合模式注重统一接口，将“一对多”的关系转化为“一对一”的关系
- 装饰者模式注重稳定接口，在此前提下为对象扩展功能
- 外观模式注重简化接口，简化组件系统与外部客户程序的依赖关系
- 享元模式注重保留接口，在内部使用共享技术对对象存储进行优化
- 代理模式注重假借接口，增加间接层来实现灵活控制

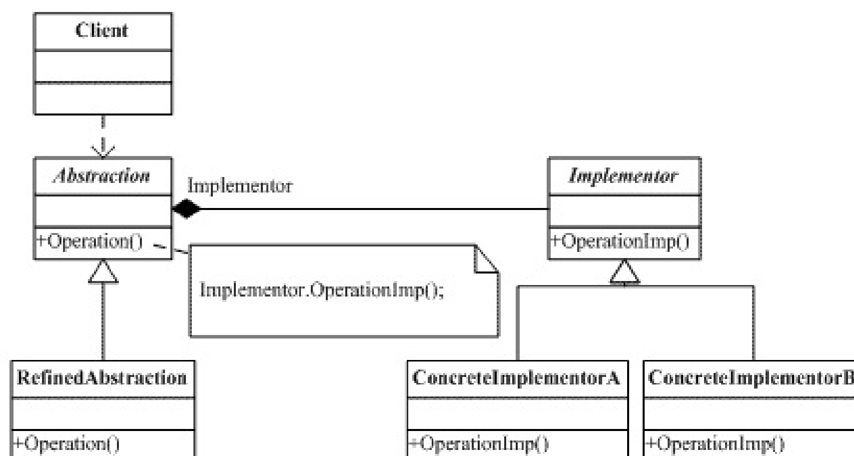
4.1 适配器模式

适配器模式意在转换接口，它能够使原本不能再一起工作的两个类一起工作，所以经常用来在类库的复用、代码迁移等方面。例如DataAdapter类就应用了适配器模式。适配器模式包括类适配器模式和对象适配器模式，具体结构如下图所示，左边是类适配器模式，右边是对象适配器模式。



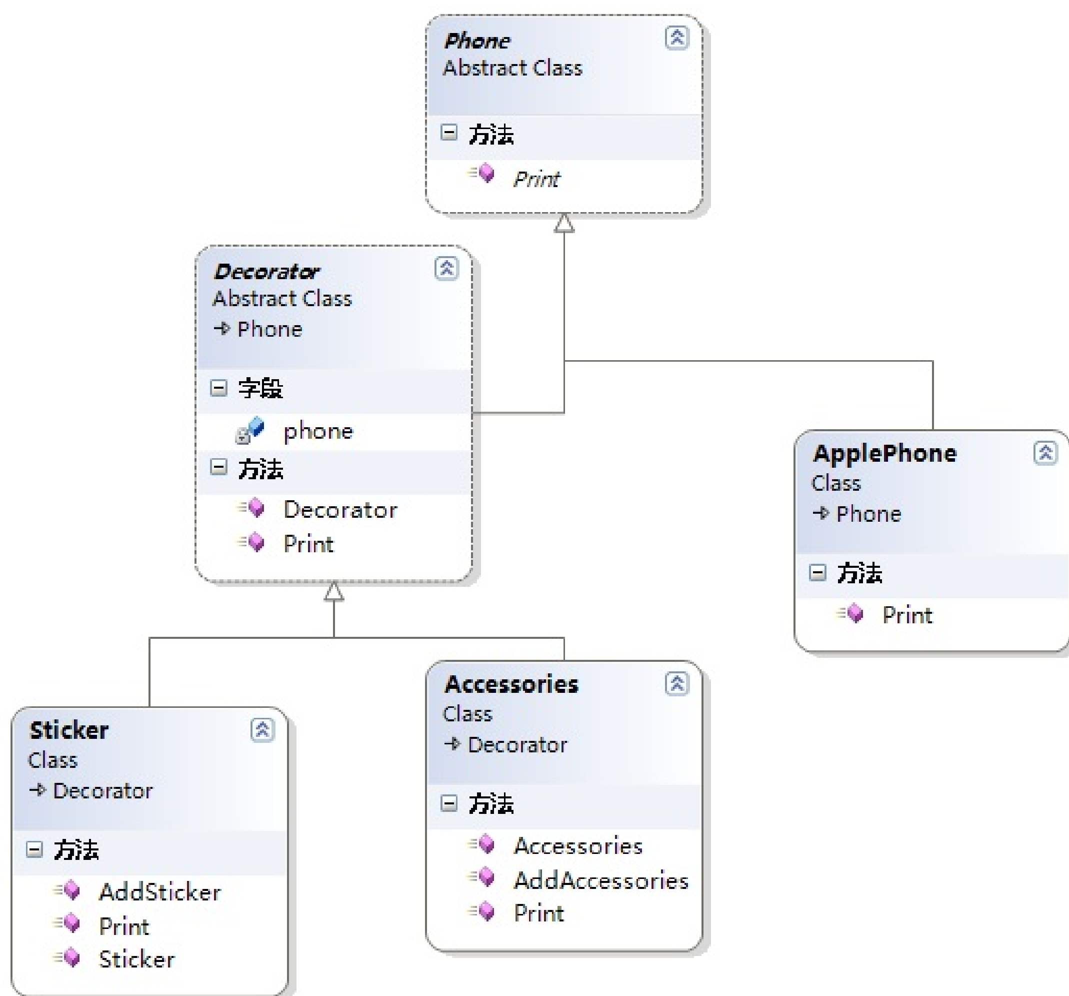
4.2 桥接模式

桥接模式旨在将抽象化与实现化解耦，使得两者可以独立地变化。意思就是说，桥接模式把原来基类的实现化细节再进一步进行抽象，构造到一个实现化的结构中，然后再把原来的基类改造成一个抽象化的等级结构，这样就可以实现系统在多个维度的独立变化，桥接模式的结构图如下所示。



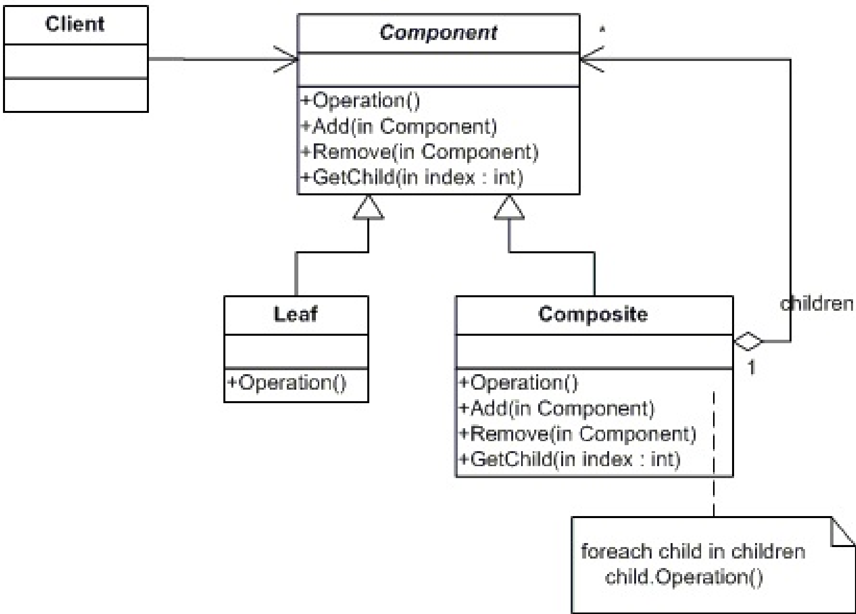
4.3 装饰者模式

装饰者模式又称包装（Wrapper）模式，它可以动态地给一个对象添加一些额外的功能，装饰者模式较继承生成子类的方式更加灵活。虽然装饰者模式能够动态地将职责附加到对象上，但它也会造成产生一些细小的对象，增加了系统的复杂度。具体的结构图如下所示。



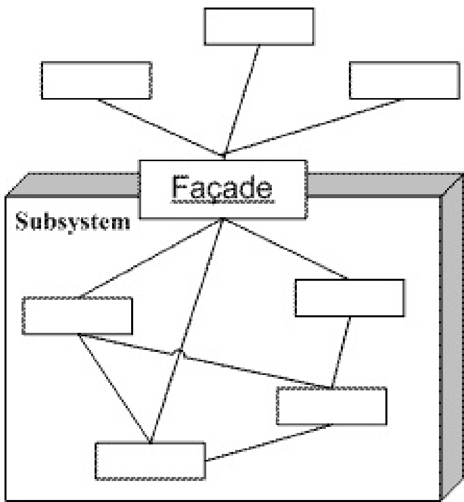
4.4 组合模式

组合模式又称为部分—整体模式。组合模式将对象组合成树形结构，用来表示整体与部分的关系。组合模式使得客户端将单个对象和组合对象同等对待。如在.NET中WinForm中的控件，TextBox、Label等简单控件继承与Control类，同时GroupBox这样的组合控件也是继承于Control类。组合模式的具体结构图如下所示。



4.5 外观模式

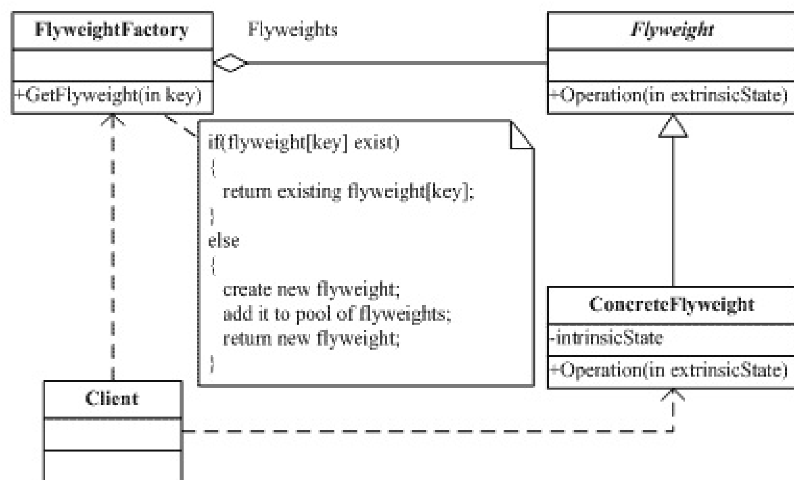
在系统中，客户端经常需要与多个子系统进行交互，这样导致客户端会随着子系统的变化而变化，此时可以使用外观模式把客户端与各个子系统解耦。外观模式指的是为子系统中的一组接口提供一个一致的门面，它提供了一个高层接口，这个接口使子系统更加容易使用。如电信的客户专员，你可以让客户专员来完成冲话费，修改套餐等业务，而不需要自己去与各个子系统进行交互。具体类结构图如下所示：



4.6 享元模式

在系统中，如何我们需要重复使用某个对象时，此时如果重复地使用new操作符来创建这个对象的话，这对系统资源是一个极大的浪费，既然每次使用的都是同一个对象，为什么不能对其共享呢？这也是享元模式出现的原因。

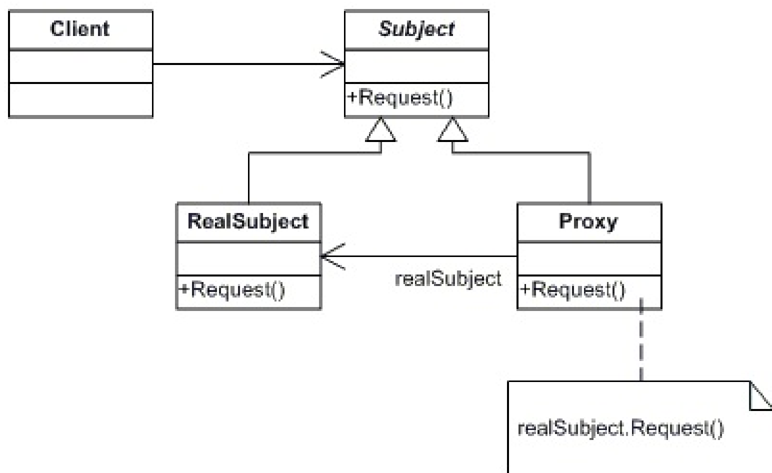
享元模式运用共享的技术有效地支持细粒度的对象，使其进行共享。在.NET类库中，String类的实现就使用了享元模式，String类采用字符串驻留池的来使字符串进行共享。更多内容参考博文：http://www.cnblogs.com/artech/archive/2010/11/25/internedstring.html。享元模式的具体结构图如下所示。



4.7 代理模式

在系统开发中，有些对象由于网络或其他的障碍，以至于不能直接对其访问，此时可以通过一个代理对象来实现对目标对象的访问。如.NET中的调用Web服务等操作。

代理模式指的是给某一个对象提供一个代理，并由代理对象控制对原对象的访问。具体的结构图如下所示。



注：外观模式、适配器模式和代理模式区别？

解答：这三个模式的相同之处是，它们都是作为客户端与真实被使用的类或系统之间的一个中间层，起到让客户端间接调用真实类的作用，不同之处在于，所应用的场合和意图不同。

代理模式与外观模式主要区别在于，代理对象无法直接访问对象，只能由代理对象提供访问，而外观对象提供对各个子系统简化访问调用接口，而适配器模式则不需要虚构一个代理者，目的是复用原有的接口。外观模式是定义新的接口，而适配器则是复用一个原有的接口。

另外，它们应用设计的不同阶段，外观模式用于设计的前期，因为系统需要前期就需要依赖于外观，而适配器应用于设计完成之后，当发现设计完成的类无法协同工作时，可以采用适配器模式。然而很多情况下在设计初期就要考虑适配器模式的使用，如涉及到大量第三方应用接口的情况；代理模式是模式完成后，想以服务的方式提供给其他客户端进行调用，此时其他客户端可以使用代理模式来对模块进行访问。

总之，代理模式提供与真实类一致的接口，旨在用来代理类来访问真实的类，外观模式旨在简化接口，适配器模式旨在转换接口。

五、行为型模式

行为型模式是对在不同对象之间划分责任和算法的抽象化。行为模式不仅仅关于类和对象，还关于它们之间的相互作用。行为型模式又分为类的行为模式和对象的行为模式两种。

- 类的行为模式——使用继承关系在几个类之间分配行为。
- 对象的行为模式——使用对象聚合的方式来分配行为。

行为型模式包括11种模式：模板方法模式、命令模式、迭代器模式、观察者模式、中介者模式、状态模式、策略模式、责任链模式、访问者模式、解释器模式和备忘录模式。

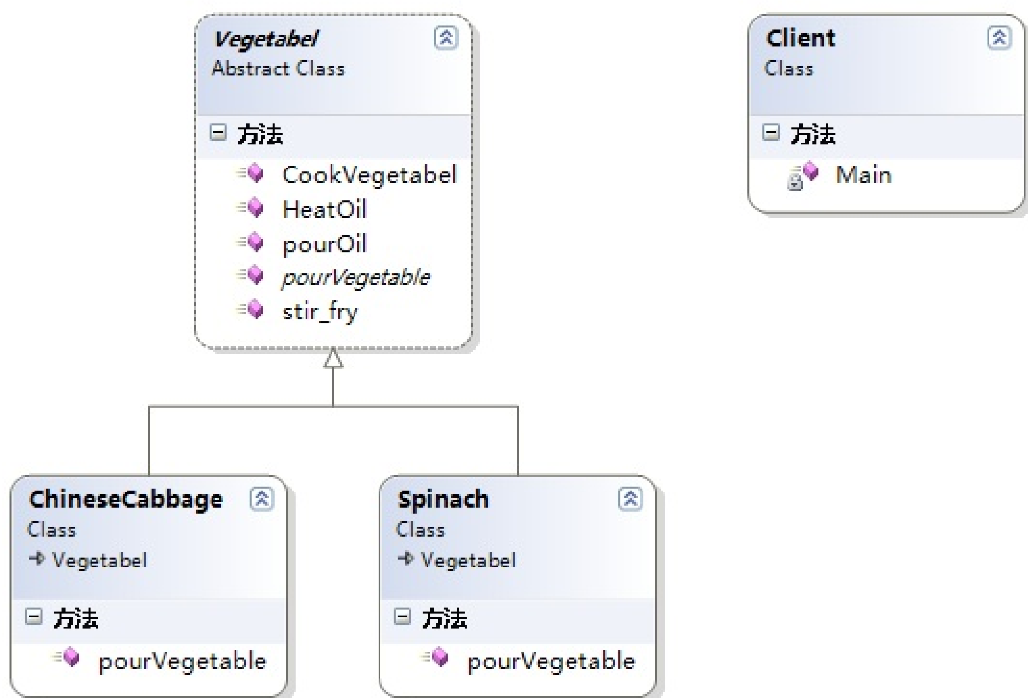
- 模板方法模式：封装算法结构，定义算法骨架，支持算法子步骤变化。
- 命令模式：注重将请求封装为对象，支持请求的变化，通过将一组行为抽象为对象，实现行为请求者和行为实现者之间的解耦。
- 迭代器模式：注重封装特定领域变化，支持集合的变化，屏蔽集合对象内部复杂结构，提供客户程序对它的透明遍历。
- 观察者模式：注重封装对象通知，支持通信对象的变化，实现对象状态改变，通知依赖它的对象并更新。
- 中介者模式：注重封装对象间的交互，通过封装一系列对象之间的复杂交互，使他们不需要显式相互引用，实现解耦。
- 状态模式：注重封装与状态相关的行为，支持状态的变化，通过封装对象状态，从而在其内部状态改变时改变它的行为。
- 策略模式：注重封装算法，支持算法的变化，通过封装一系列算法，从而可以随时独立于客户替换算法。
- 责任链模式：注重封装对象责任，支持责任的变化，通过动态构建职责链，实现事务处理。
- 访问者模式：注重封装对象操作变化，支持在运行时为类结构添加新的操作，在类层次结构中，在不改变各类的前提下定义作用于这些类实例的新的操作。
- 备忘录模式：注重封装对象状态变化，支持状态保存、恢复。
- 解释器模式：注重封装特定领域变化，支持领域问题的频繁变化，将特定领域的问题表达为某种语法规则下的句子，然后构建一个解释器来解释这样的句

子，从而达到解决问题的目的。

5.1 模板方法模式

在现实生活中，有论文模板，简历模板等。在现实生活中，模板的概念是给定一定的格式，然后其他所有使用模板的人可以根据自己的需求去实现它。同样，模板方法也是这样的。

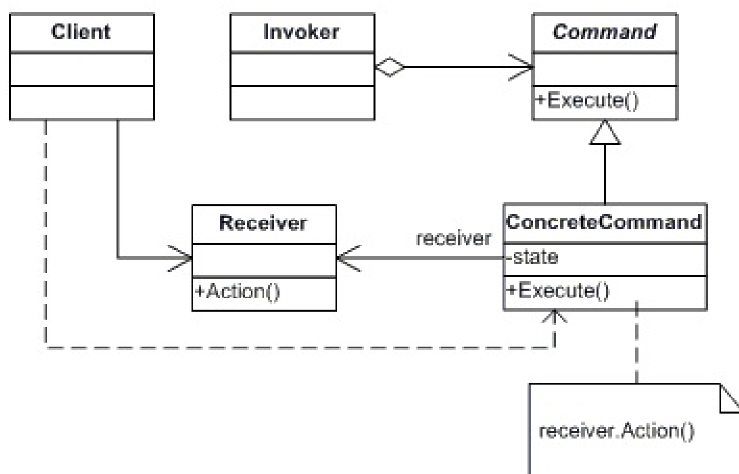
模板方法模式是在一个抽象类中定义一个操作中的算法骨架，而将一些具体步骤实现延迟到子类中去实现。模板方法使得子类可以不改变算法结构的前提下，重新定义算法的特定步骤，从而达到复用代码的效果。具体的结构图如下所示。



以生活中做菜为例子实现的模板方法结构图

5.2 命令模式

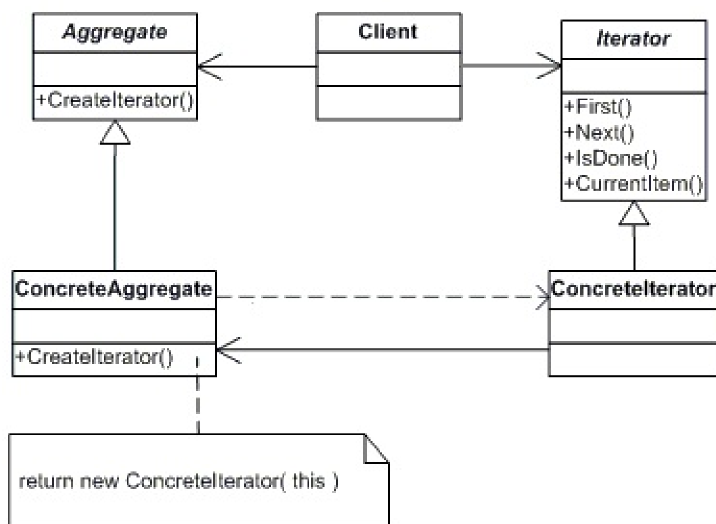
命令模式属于对象的行为模式，命令模式把一个请求或操作封装到一个对象中，通过对命令的抽象化来使得发出命令的责任和执行命令的责任分隔开。命令模式的实现可以提供命令的撤销和恢复功能。具体的结构图如下所示。



5.3 迭代器模式

迭代器模式是针对集合对象而生的，对于集合对象而言，必然涉及到集合元素的添加删除操作，也肯定支持遍历集合元素的操作，此时如果把遍历操作也放在集合对象的话，集合对象就承担太多的责任了，此时可以进行责任分离，把集合的遍历放在另一个对象中，这个对象就是迭代器对象。

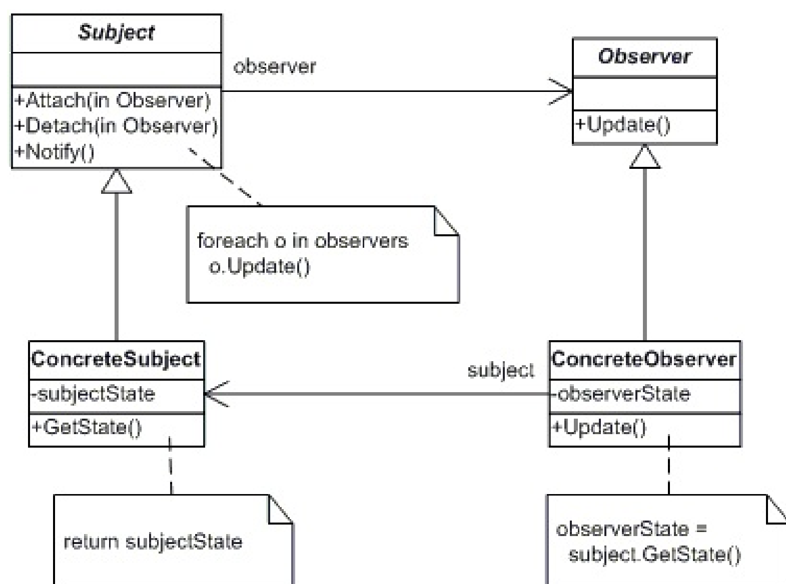
迭代器模式提供了一种方法来顺序访问一个集合对象中各个元素，而又无需暴露该对象的内部表示，这样既可以做到不暴露集合的内部结构，又可以让外部代码透明地访问集合内部元素。具体的结构图如下所示。



5.4 观察者模式

在现实生活中，处处可见观察者模式，例如，微信中的订阅号，订阅博客和QQ微博中关注好友，这些都属于观察者模式的应用。

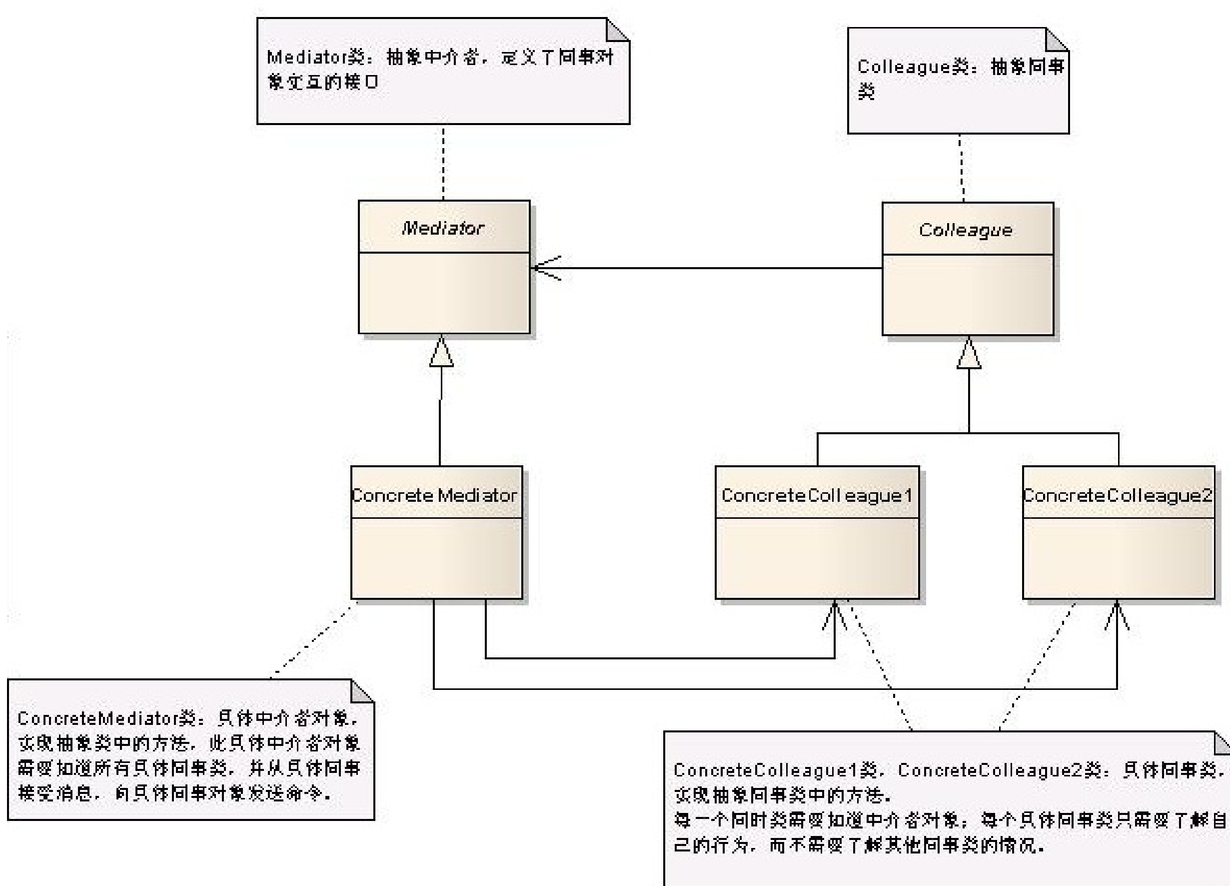
观察者模式定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象，这个主题对象在状态发生变化时，会通知所有观察者对象，使它们能够自动更新自己的行为。具体结构图如下所示：



5.5 中介者模式

在现实生活中，有很多中介者模式的身影，例如QQ游戏平台，聊天室、QQ群和短信平台，这些都是中介者模式在现实生活中的应用。

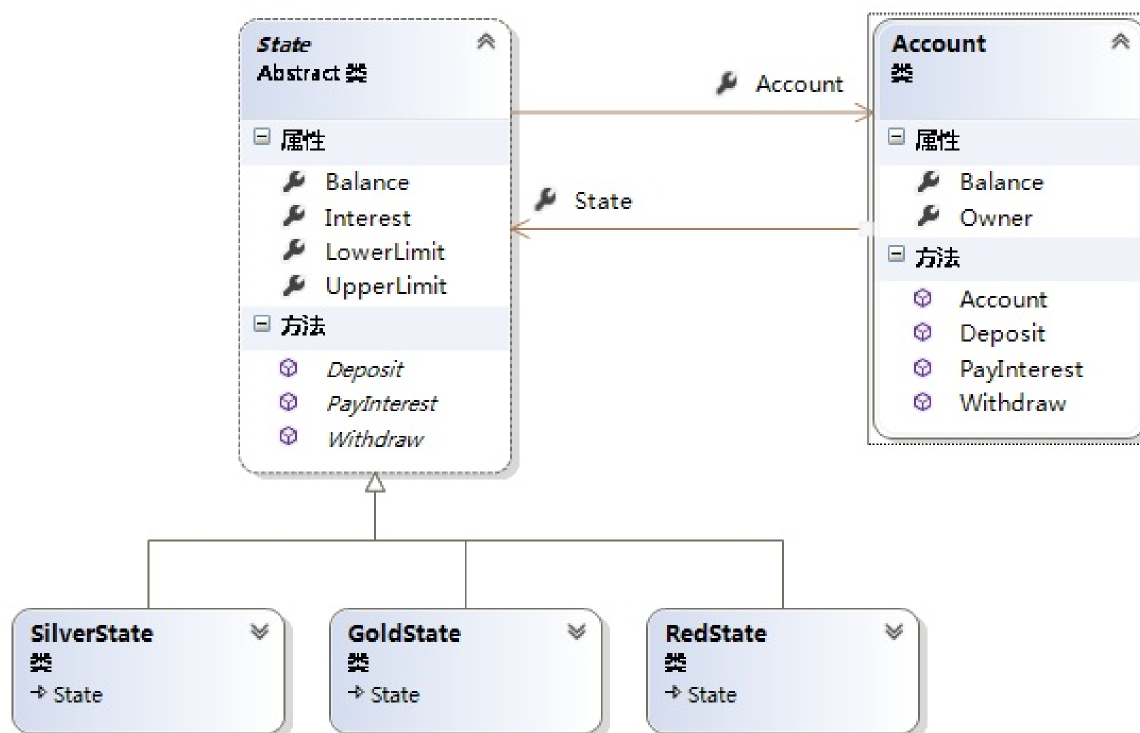
中介者模式，定义了一个中介对象来封装一系列对象之间的交互关系。中介者使各个对象之间不需要显式地相互引用，从而使耦合性降低，而且可以独立地改变它们之间的交互行为。具体的结构图如下所示：



5.6 状态模式

每个对象都有其对应的状态，而每个状态又对应一些相应的行为，如果某个对象有多个状态时，那么就会对应很多的行为。那么对这些状态的判断和根据状态完成的行为，就会导致多重条件语句，并且如果添加一种新的状态时，需要更改之前现有的代码。这样的设计显然违背了开闭原则，状态模式正是用来解决这样的问题的。

状态模式——允许一个对象在其内部状态改变时自动改变其行为，对象看起来就像是改变了它的类。具体的结构图如下所示：

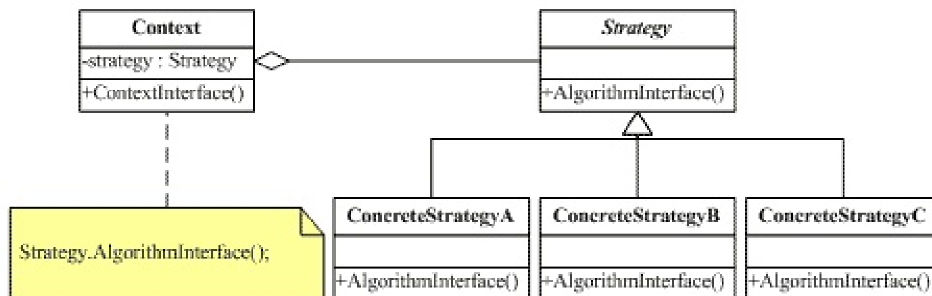


5.7 策略模式

在现实生活中，中国的所得税，分为企业所得税、外商投资企业或外商企业所得税和个人所得税，针对于这3种所得税，每种所计算的方式不同，个人所得税有个人所得税的计算方式，而企业所得税有其对应计算方式。如果不采用策略模式来实现这样一个需求的话，我们会定义一个所得税类，该类有一个属性来标识所得税的类型，并且有一个计算税收的CalculateTax()方法，在该方法体内需要对税收类型进行判断，通过if-else语句来针对不同的税收类型来计算其所得税。这样的实现确实可以解决这个场景，但是这样的设计不利于扩展，如果系统后期需要增加一种所得税时，此时不得不回去修改CalculateTax方法来多添加一个判断语句，这样明白违背了“开放——封闭”原则。此时，我们可以考虑使用策略模式来解决这个问题，既然税收方法是这个场景中的变化部分，此时自然可以想到对税收方法进行抽象，这也是策略模式实现的精髓所在。

策略模式是对算法的包装，是把使用算法的责任和算法本身分割开，委派给不同的对象负责。策略模式通常把一系列的算法包装到一系列的策略类里面。用一句话概括策略模式就是——“将每个算法封装到不同的策略类中，使得它们可以互换”。下

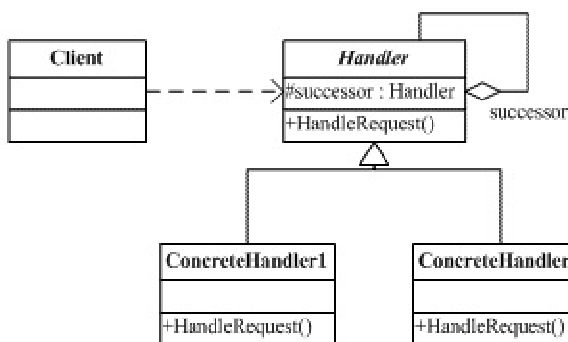
面是策略模式的结构图：



5.8 责任链模式

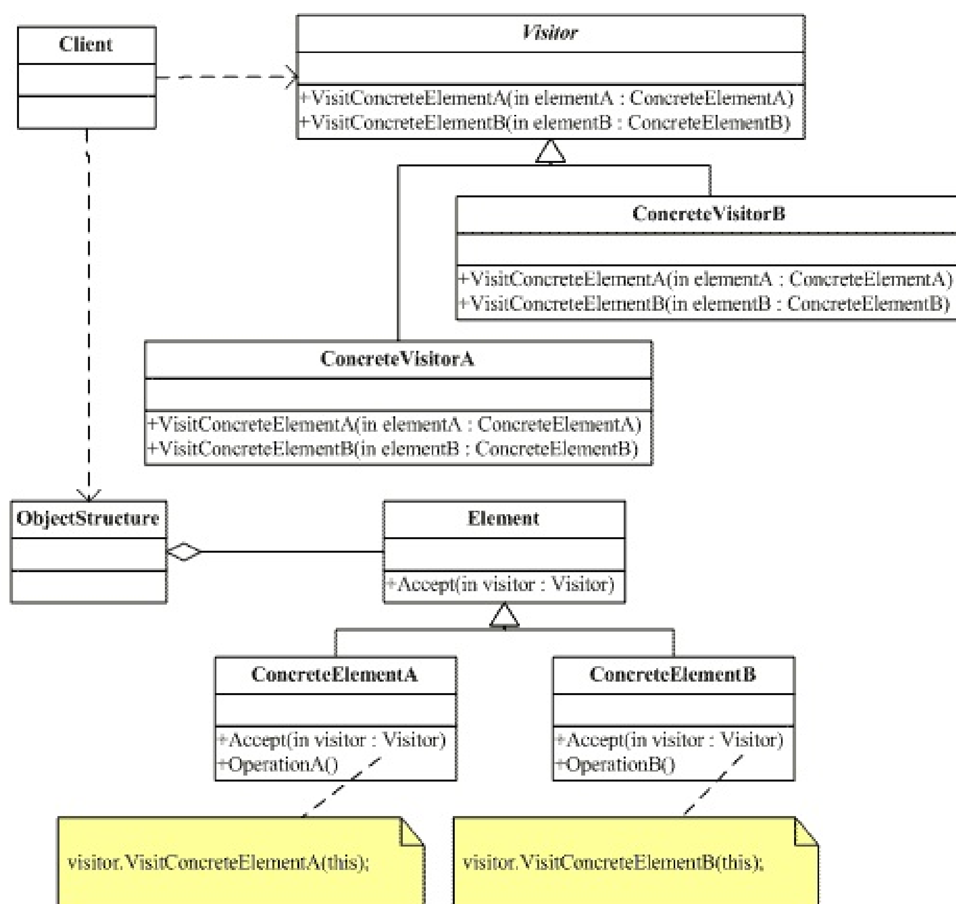
在现实生活中，有很多请求并不是一个人说了就算的，例如面试时的工资，低于1万的薪水可能技术经理就可以决定了，但是1万~1万5的薪水可能技术经理就没这个权利批准，可能需要请求技术总监的批准。

责任链模式——某个请求需要多个对象进行处理，从而避免请求的发送者和接收之间的耦合关系。将这些对象连成一条链子，并沿着这条链子传递该请求，直到有对象处理它为止。具体结构图如下所示：



5.9 访问者模式

访问者模式是封装一些施加于某种数据结构之上的操作。一旦这些操作需要修改的话，接受这个操作的数据结构则可以保存不变。访问者模式适用于数据结构相对稳定的系统， 它把数据结构和作用于数据结构之上的操作之间的耦合度降低，使得操作集合可以相对自由地改变。具体结构图如下所示：



5.10 备忘录模式

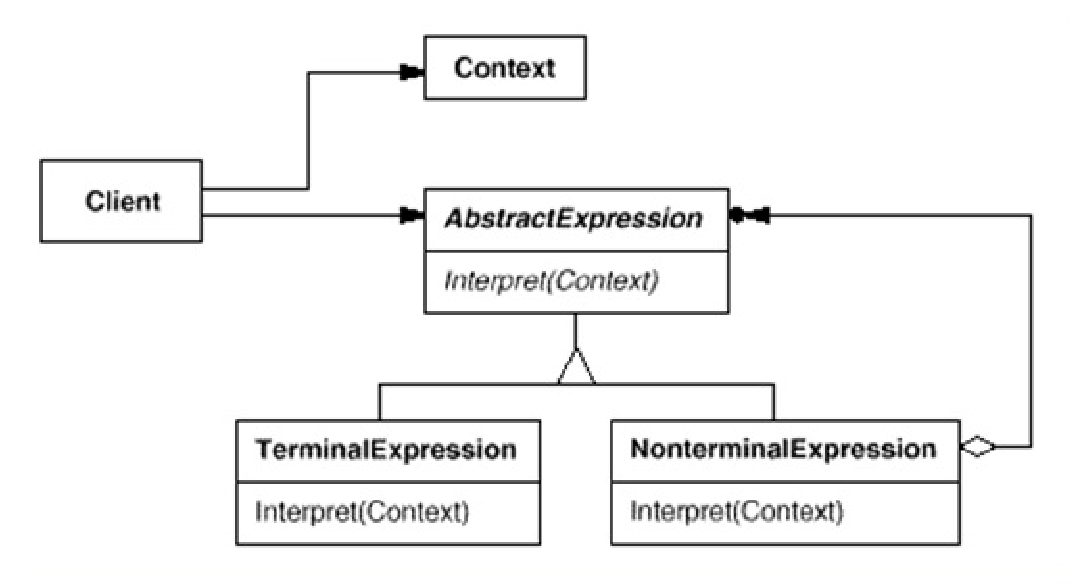
生活中的手机通讯录备忘录，操作系统备份点，数据库备份等都是备忘录模式的应用。备忘录模式是在不破坏封装的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，这样以后就可以把该对象恢复到原先的状态。具体的结构图如下所示：



5.11 解释器模式

解释器模式是一个比较少用的模式，所以我自己也没有对该模式进行深入研究，在生活中，英汉词典的作用就是实现英文和中文互译，这就是解释器模式的应用。

解释器模式是给定一种语言，定义它文法的一种表示，并定义一种解释器，这个解释器使用该表示来解释器语言中的句子。具体的结构图如下所示：



六、总结

23种设计模式，其实前辈们总结出来解决问题的方式，它们追求的宗旨还是保证系统的低耦合高内聚，指导它们的原则无非就是封装变化，责任单一，面向接口编程等设计原则。之后，我会继续分享自己WCF的学习过程，尽管博客园中有很多WCF系列，之前觉得没必要写，觉得会用就行了，但是不写，总感觉知识不是自己的，感觉没有深入，所以还是想写这样一个系列，希望各位博友后面多多支持。

PS：很多论坛都看到初学者问，WCF现在还有没有必要深入学之类的问题，因为他们觉得这些技术可能会过时，说不定到时候微软又推出了一个新的SOA的实现方案了，那岂不是白花时间深入学了，所以就觉得没必要深入去学，知道用就可以了。对于这个问题，我之前也有这样同样的感觉，但是现在我觉得，尽管WCF技术可能会被替换，但深入了解一门技术，重点不是知道一些更高深API的调用啊，而是了解它的实现机制和思维方式，即使后面这个技术被替代了，其背后机制也肯定是相似的。所以深入了解了一个技术，你就会感觉新的技术熟悉，对其感觉放松。并且，你深入了解完一门技术之后，你面试时也敢说你很好掌握了这门技术，而不至于说平时使用的很多，一旦深入问时却不知道背后实现原理。这也是我要写WCF系列的原因。希望这点意见对一些初学者有帮助。

WPF快速入门系列

WPF快速入门系列(1)——WPF布局概览

一、引言

关于WPF早在一年前就已经看过《深入浅出WPF》这本书，当时看完之后由于没有做笔记，以至于我现在又重新捡起来并记录下学习的过程，本系列将是一个WPF快速入门系列，主要介绍WPF中主要的几个不同的特性，如依赖属性、命令、路由事件等。

在正式介绍之前，我还想分享下为什么我又要重新捡起来WPF呢？之前没有记录下来的原来主要是打算走互联网方向的，后面发现互联网方向经常加班，又累，有时候忙的连自己写了什么都不知道的，所以后面机缘巧合地进了一家外企，在外企不像互联网行业那样，比较清楚，有更多的时间去理清楚自己所学习到的知识，其中同时也发现了WPF的重要性和应用场景，在一些美资企业和印度的公司，客户端都非常喜欢用WPF来做演示的客户端，所以，自然走上外企这条路，所以就打算好好研究下WPF了，所以也就有了这个系列。

二、WPF的自我介绍

Windows Presentation Foudation,WPF是下一代显示系统，用来生成能带给用户震撼视觉体验的Windows客户端应用程序。WPF的核心是一个与分辨率无关并且基于向量的程序引擎，目的在于利用现代图形硬件的优势。WPF在.NET Framework 3.0中被微软引入到.NET Framework类库中，并且在.NET 3.5、4.0 和4.5都有所更新。WPF可以理解为是实现下一代Windows 桌面应用程序的技术，在之前我们通常会使用MFC或Winform来实现Windows桌面程序。

WPF除了引入了新的API之前，还引入了一些新的概念，这些新的概念会在本系列中一一介绍。众所周知，在实现桌面应用程序之前，第一步必然是对窗体进行布局，WPF为了更好地实现布局，提供了很多布局控件，下面就让我们一起去看看WPF布局组件。

三、WPF布局详解

WPF的布局控件都继承于System.Windows.Controls.Panel这个类，本文主要介绍在Panel基类下的几个常用的布局控件。下图是布局控件的继承关系：

System.Windows.Controls.Panel
  System.Windows.Controls.Canvas
  System.Windows.Controls.DockPanel
  System.Windows.Controls.Grid
  System.Windows.Controls.Primitives.TabPanel
  System.Windows.Controls.Primitives.ToolBarOverflowPanel
  System.Windows.Controls.Primitives.UniformGrid
  System.Windows.Controls.Ribbon.Primitives.RibbonContextualTabGroupsPanel
  System.Windows.Controls.Ribbon.Primitives.RibbonGalleryCategoriesPanel
  System.Windows.Controls.Ribbon.Primitives.RibbonGalleryItemsPanel
  System.Windows.Controls.Ribbon.Primitives.RibbonGroupItemsPanel
  System.Windows.Controls.Ribbon.Primitives.RibbonQuickAccessToolBarOverflowPanel
  System.Windows.Controls.Ribbon.Primitives.RibbonTabHeadersPanel
  System.Windows.Controls.Ribbon.Primitives.RibbonTabsPanel
  System.Windows.Controls.Ribbon.Primitives.RibbonTitlePanel
  System.Windows.Controls.StackPanel
  System.Windows.Controls.VirtualizingPanel
  System.Windows.Controls.WrapPanel

3.1 WPF布局过程

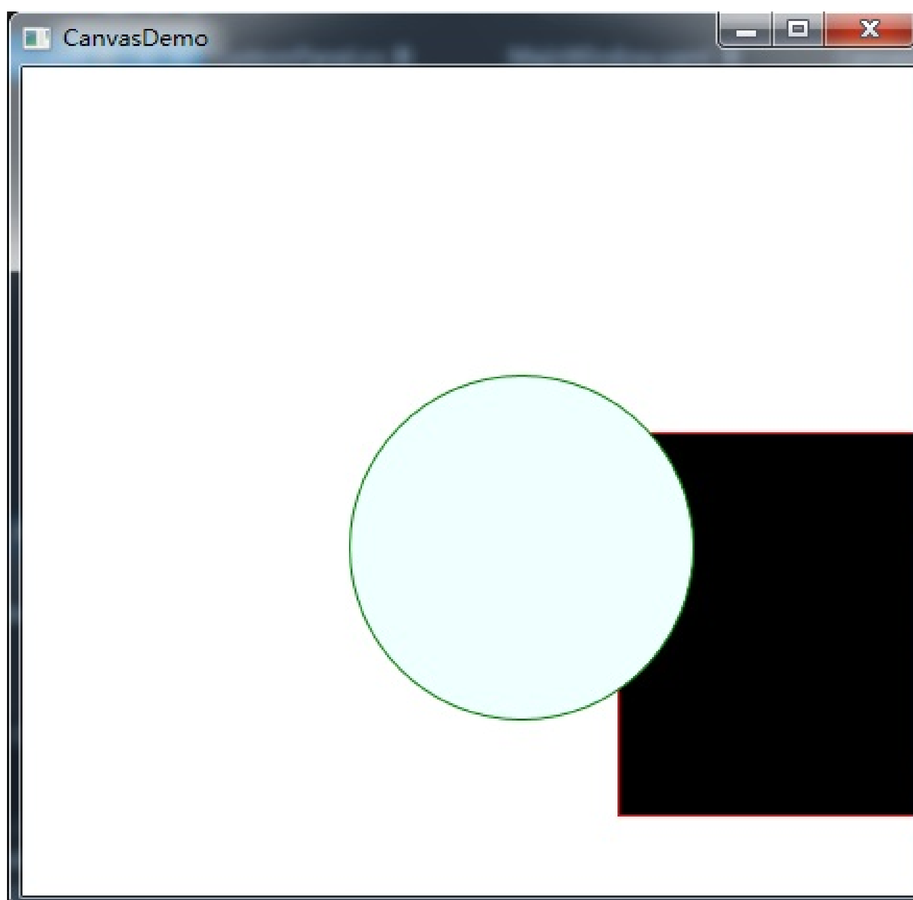
WPF布局包括两个阶段：一个测量（**measure**）阶段和一个排列**(arrange)**阶段。在测量阶段，容器遍历所有子元素，并询问子元素它们所期望的大小。在排列阶段，容器在合适的位置放置子元素。WPF布局可以理解为一个递归过程，它会递归对布局控件内的每个子元素进行大小调整，定位和绘制，最后进行呈现，直到递归所有子元素为止，这样也就完成了整个布局过程。

布局系统为每个子元素完成了两个处理过程：测量处理和排列处理。每个Panel都提供了自己的**MeasureOverride.aspx)**和**ArrangeOverride.aspx)**方法，以实现自己特定的布局行为。所以，你如果想自定义布局控件，也可以重新这两个方法来达到，关于自定义布局控件会在后面介绍到。

3.2 Canvas 布局控件

Canvas面板是最轻量级的布局容器，它不会自动调整内部元素的排列和大小，不指定元素位置，元素将默认显示在画布的左上方。Canvas主要用来画图。Canvas默认不会自动裁剪超过自身范围的内容，即溢出的内容会显示在Canvas外面，这是因为Canvas的ClipToBounds属性默认值是false，我们可以显式地设置为true来裁剪多出的内容。下面XAML代码简单演示了Canvas面板的使用。

上面XAML实现的效果如下图所示：



其中，矩形的右边区域以溢出Canvas面板区域，如向右拉动边框，此时Canvas会拉伸以填满可用空间，此时就可以看到矩形溢出的部分。但Canvas面板内的控件不会改变其尺寸和位置。对应的C#代码实现如下所示：

```
 1 public partial class CanvasDemo : Window
 2     {
 3         public CanvasDemo()
 4         {
 5             InitializeComponent();
 6 
 7             Canvas canv = new Canvas();
 8             canv.Margin = new Thickness(10, 10, 10, 10);
 9             canv.Background = new SolidColorBrush(Colors.White);
10 
11             // 把canv添加为窗体的子控件
12             this.Content = canv;
13 
14             // Rectangle
15             Rectangle rect = new Rectangle();
16             rect.Fill = new SolidColorBrush(Colors.Black);
17             rect.Stroke = new SolidColorBrush(Colors.Red);
18             rect.Width = 200;
19             rect.Height = 200;
20             rect.SetValue(Canvas.LeftProperty, (double)300);
21             rect.SetValue(Canvas.TopProperty, (double)180);
22             canv.Children.Add(rect);
23 
24             // Ellipse
25             Ellipse el = new Ellipse();
26             el.Fill = new SolidColorBrush(Colors.Azure);
27             el.Stroke = new SolidColorBrush(Colors.Green);
28             el.Width = 180;
29             el.Height = 180;
30             el.SetValue(Canvas.LeftProperty, (double)160);
31             // 必须转换为double，否则执行会出现异常
32             // 详细介绍见：http://msdn.microsoft.com/zh-cn/library
33             el.SetValue(Canvas.TopProperty, (double)150);
34             el.SetValue(Panel.ZIndexProperty, -1);
35             canv.Children.Add(el);
36 
37             // Print Zindex Value
38             int zRectIndex = (int)rect.GetValue(Panel.ZIndexProp
39             int zelIndex = (int)el.GetValue(Panel.ZIndexProperty
40             Debug.WriteLine("Rect ZIndex is: {0}", zRectIndex);
41             Debug.WriteLine("Ellipse ZIndex is: {0}", zelIndex);
42         }
43     }
```

从上面可以看出，即使C#代码可以实现完全一样的效果，但是需要书写更多的代码，所以，在平时开发中，对于控件的布局，一般采用XAML的方式，C#代码一般用于在运行时加载某个控件到界面中的实现。

3.3 StackPanel 布局控件

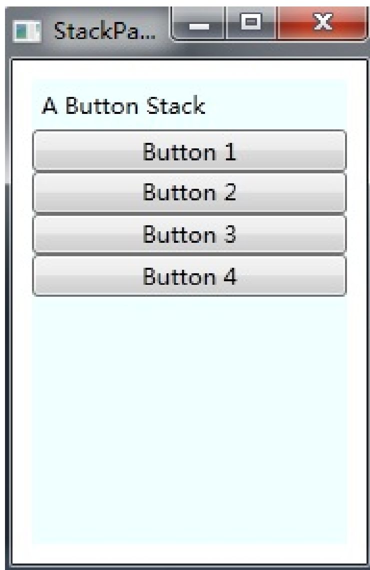
StackPanel就是将子元素按照堆栈的形式一一排列，可以通过设置StackPanel的Orientation属性设置两种排列方式：横排（Horizontal，该值为默认值）和竖排（Vertical）。纵向的StackPanel每个元素默认宽度与面板一样宽，反之横向是高度和面板一样高。如果包含的元素超过了面板控件，它会被截断多出的内容。可以通过Orientation属性来设置StackPanel是横排（设置其值为Vertical）还是竖排（设置其值为Horizontal）。下面XAML代码演示了StackPanel的使用：

```
 1 <Window x:Class="WPFLayoutDemo.StackPanelDemo"
 2         xmlns="http://schemas.microsoft.com/winfx/2006/xaml/pres
 3         xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 4         Title="StackPanel" Height="300" Width="200">
 5     <StackPanel Margin="10,10,10,10" Background="Azure">
 6         <Label>A Button Stack</Label>
 7         <Button Content="Button 1"></Button>
 8         <Button>Button 2</Button>
 9         <Button>Button 3</Button>
10         <Button Content="Button 4"></Button>
11     </StackPanel>
12 </Window>
```

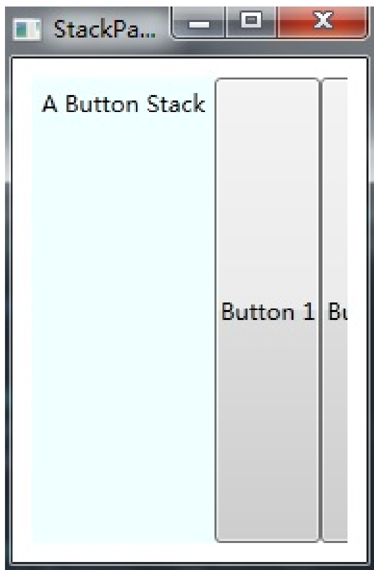
对应的C#实现代码如下所示：

```
1 public partial class StackPanelDemo : Window
2     {
3         public StackPanelDemo()
4         {
5             InitializeComponent();
6             StackPanel sp = new StackPanel();
7             sp.Margin = new Thickness(10, 10, 10, 10);
8             sp.Background = new SolidColorBrush(Colors.Azure);
9             sp.Orientation = Orientation.Vertical;
10            // 把sp添加为窗体的子控件
11            this.Content = sp;
12
13            // Label
14            Label lb = new Label();
15            lb.Content = "A Button Stack";
16            sp.Children.Add(lb);
17
18            // Button 1
19            Button btn1 = new Button();
20            btn1.Content = "Button 1";
21            sp.Children.Add(btn1);
22
23            // Button 2
24            Button btn2 = new Button();
25            btn2.Content = "Button 2";
26            sp.Children.Add(btn2);
27
28            // Button 3
29            Button btn3 = new Button();
30            btn3.Content = "Button 3";
31            sp.Children.Add(btn3);
32
33            // Button 4
34            Button btn4 = new Button();
35            btn4.Content = "Button 4";
36            sp.Children.Add(btn4);
37        }
38    }
```

上面代码的实现效果如下图所示：



如果将StackPanel的Orientation属性设置为“Horizontal”的话，此时的效果如下图所示：



尽管布局由容器决定，但子元素仍然有一定的决定权，布局面板支持一些布局属性，以便与子元素结合使用，在下图中列出了这些布局属性：

名 称	说 明
HorizontalAlignment	当水平方向上有额外的空间时，该属性决定了子元素在布局容器中如何定位。可以选择使用 Center、Left、Right 或 Stretch 等属性值
VerticalAlignment	当垂直方向上有额外的空间时，该属性决定了子元素在布局容器控件中如何定位。可以选择使用 Center、Top、Bottom 或 Stretch 等属性值
Margin	该属性用于在元素的周围添加一定的空间。Margin 属性是 System.Windows.Thickness 结构的一个实例，该结构具有分别用于为顶部、底部、左边和右边添加空间的独立组件
MinWidth 和 MinHeight	这两个属性用于设置元素的最小尺寸。如果一个元素对于其他布局容器来说太大，该元素将被剪裁以适合容器
MaxWidth 和 MaxHeight	这两个属性用于设置元素的最大尺寸。如果有更多可以使用的空间，那么在扩展子元素时就不会超出这一限制，即使把 HorizontalAlignment 和 Vertical Alignment 属性设置为 Stretch 也同样如此
Width 和 Height	这两个属性用于显式地设置元素的尺寸。这一设置会覆盖为 HorizontalAlignment 属性和 VerticalAlignment 属性设置的 Stretch 值。但是不能超出 MinWidth、MinHeight、MaxWidth 和 MaxHeight 属性设置的范围

3.4 WrapPanel 布局控件

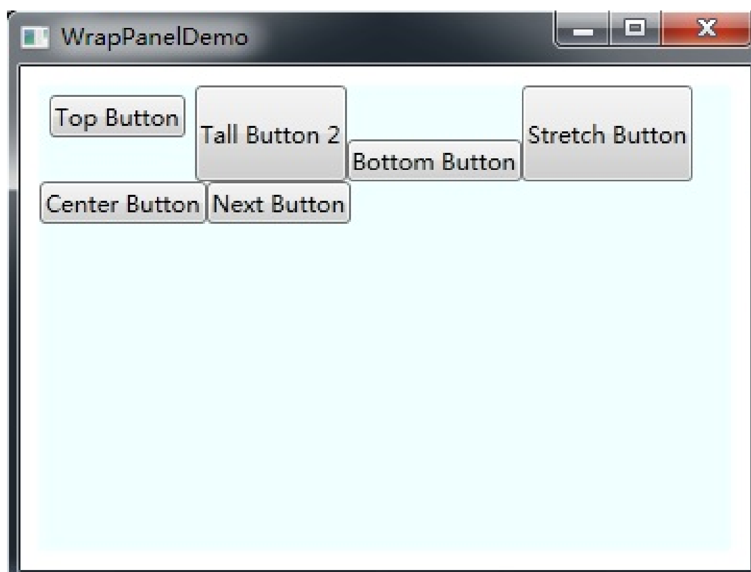
WrapPanel面板在可能的空间中，一次以一行或一列的方式布置控件。默认情况下，WrapPanel.Orientation属性设置为Horizontal，控件从左向右进行排列，然后再在下一行中排列，但你可将WrapPanel.Orientation设置为Vertical，从而在多个列中放置元素。与StackPanel面板不同，WrapPanel面板实际上用来控制用户界面中一小部分的布局细节，并非用于控制整个窗口布局。

下面示例中定义了一系列具有不同对齐方式的按钮，并将这些按钮放在一个WrapPanel面板中。

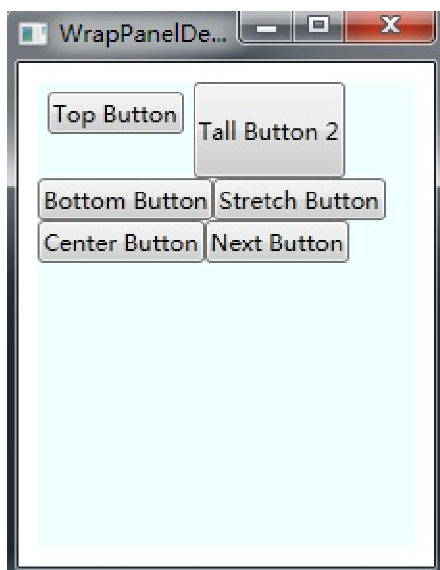
```
 1 <Window x:Class="WPFLayoutDemo.WrapPanelDemo"
 2         xmlns="http://schemas.microsoft.com/winfx/2006/xaml/pres
 3         xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 4         Title="WrapPanelDemo" Height="300" Width="500">
 5     <WrapPanel Margin="10" Background="Azure">
 6         <Button VerticalAlignment="Top" Margin="5">Top Button</B
 7         <Button MinHeight="50"> Tall Button 2</Button>
 8         <Button VerticalAlignment="Bottom">Bottom Button</Button
 9         <Button>Stretch Button</Button>
10         <Button VerticalAlignment="Center">Center Button</Button
11         <Button>Next Button</Button>
12     </WrapPanel>
13 </Window>
```

下图显示了如何对这些按钮进行换行以适应WrapPanel面板的当前尺寸，WrapPanel面板的当前尺寸由包含它的窗口尺寸决定的。在上面的例子中，WrapPanel面板水平地创建一系列假象的行，每一行的搞定都被设置为所包含元素

中最高元素的高度。其他空间可能被拉伸以适应该高度，或根据VerticalAlignment属性设置进行对齐。



当缩小窗口大小时，对应的WrapPanel也会改变，从而改变WrapPanel面板中控件的排列，具体效果如下图所示：



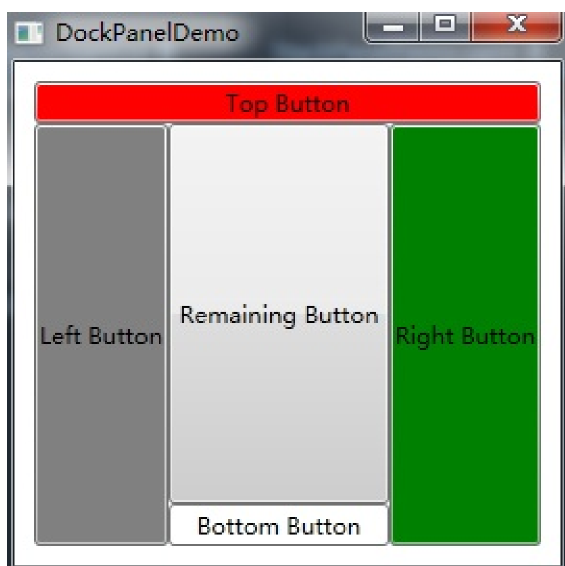
3.5 DockPanel 布局控件

DockPanel面板定义一个区域，在此区域中，你可以使子元素通过锚点的形式进行排列。DockPanel类似于WinForm中Dock属性的功能。对于在DockPanel中的元素的停靠可以通过Panel.Dock的附加属性来设置，如果设置LastChildFill属性为true，则最后一个元素将填充剩余的所有空间。

下面XAML代码演示了DockPanel控件的使用：

```
 1 <Window x:Class="WPFLayoutDemo.DockPanelDemo"
 2         xmlns="http://schemas.microsoft.com/winfx/2006/xaml/pres
 3         xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 4         Title="DockPanelDemo" Height="300" Width="300">
 5     <DockPanel Margin="10" Background="Azure" LastChildFill="Tru
 6         <Button DockPanel.Dock="Top" Background="Red">Top Button
 7         <Button DockPanel.Dock="Left" Background="Gray">Left But
 8         <Button DockPanel.Dock="Right" Background="Green">Right
 9         <Button DockPanel.Dock="Bottom"  Background="White">Bott
10         <Button>Remaining Button</Button>
11     </DockPanel>
12 </Window>
```

运行的效果如下图所示：



3.6 Grid 布局控件

Grid比起其他Panel，功能是最多最为复杂的布局控件。它由<Grid.ColumnDefinitions>列元素集合和<Grid.RowDefinitions>行元素集合两种元素组成。而放在Grid面板中的元素必须显式采用附加属性定义其所在行和列，否则元素均默认放置在第0行第0列。下面XAML演示了Grid面板的使用：

```
1 <Window x:Class="WPFLayoutDemo.GridDemo"
2         xmlns="http://schemas.microsoft.com/winfx/2006/xaml/pres
3         xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4         Title="GridDemo" Height="300" Width="480">
5     <Grid Width="Auto" Height="Auto">
6         <Grid.RowDefinitions>
7             <RowDefinition Height="*"/>
8             <RowDefinition Height="Auto"/>
9         </Grid.RowDefinitions>
10        <Grid.ColumnDefinitions>
11            <ColumnDefinition Width="120"/>
12            <ColumnDefinition Width="150"/>
13            <ColumnDefinition Width="*"/>
14            <ColumnDefinition Width="2*"/>
15        </Grid.ColumnDefinitions>
16        <Rectangle Grid.Row="0" Grid.Column="0" Fill="Green" Ma
17        <Rectangle Grid.Row="0" Grid.Column="1" Grid.ColumnSpan=
18        <Rectangle Grid.Row="0" Grid.Column="4" Fill="Orange"/>
19        <Button Grid.Row="1" Grid.Column="0">Button 2</Button>
20        <Rectangle Grid.Row="1" Grid.Column="1" Grid.ColumnSpan=
21    </Grid>
22 </Window>
```

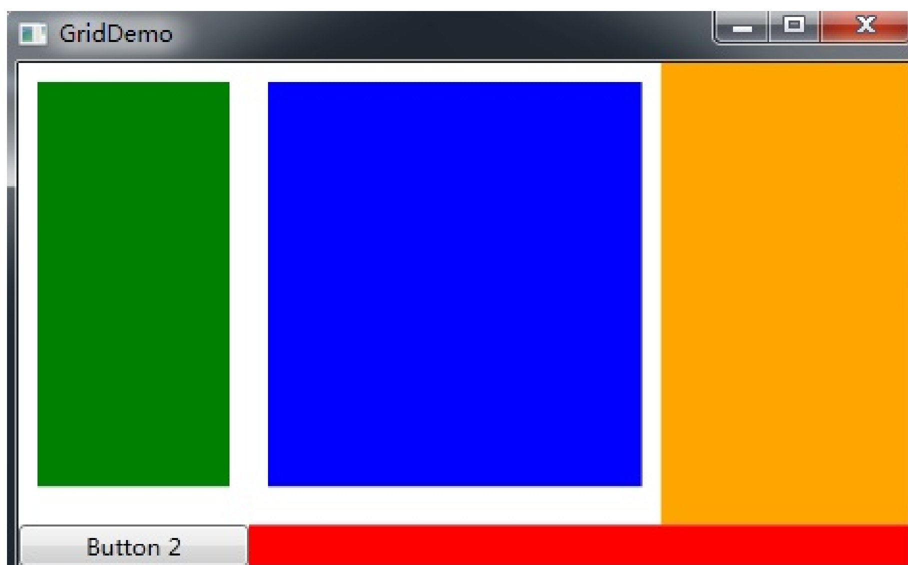
定义Grid的列宽和行高可采用固定、自动和按比例三种方式定义。

第一种：固定长度——宽度不够时，元素会被裁剪，单位是pixel;

第二种：自动长度——自动匹配行中最宽元素的高度。

第三种：比例长度——""表示占用剩余的全部宽度或高度，两行都是，则将剩余高度平分。像上面的一个2，一个，表示前者2/3宽度。

其运行效果如下图所示：



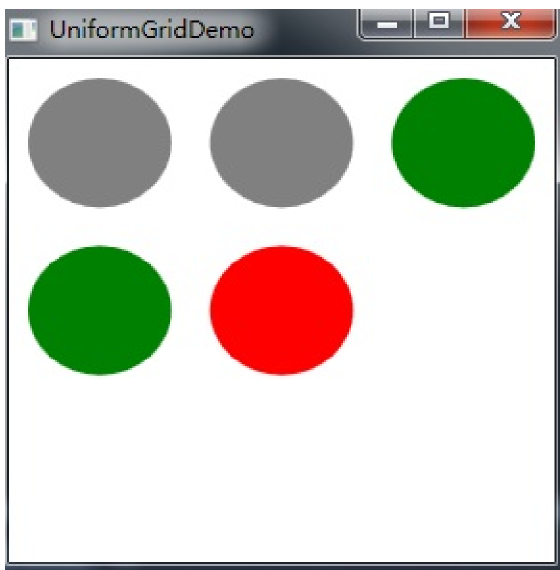
3.7 UniformGrid 布局控件

UniformGrid是Grid简化版本，不像Grid面板，UniformGrid不需要预先定义行集合和列集合，反而，通过简单设置Rows和Columns属性来设置尺寸。每个单元格始终具有相同的大小。UniformGrid每个单元格只能容纳一个元素，将自动按照在其内部的元素个数，自动创建行和列，并通过保存相同的行列数。

下面XAML演示了UniformGrid控件的使用：

```
 1 <Window x:Class="WPFLayoutDemo.UniformGridDemo"
 2         xmlns="http://schemas.microsoft.com/winfx/2006/xaml/pres
 3         xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 4         Title="UniformGridDemo" Height="300" Width="300">
 5     <UniformGrid>
 6         <Ellipse Margin="10" Fill="Gray"/>
 7         <Ellipse Margin="10" Fill="Gray"/>
 8         <Ellipse Margin="10" Fill="Green"/>
 9         <Ellipse Margin="10" Fill="Green"/>
10         <Ellipse Margin="10" Fill="Red"/>
11     </UniformGrid>
12 </Window>
```

在上面，并没有显示指定UniformGrid的行和列数，此时UniformGrid将自动按照元素的个数，自动创建行和列。运行效果如下图所示。最好是显式指定Rows和Columns属性，这样才能确保布局是按照你的思路去进行的。

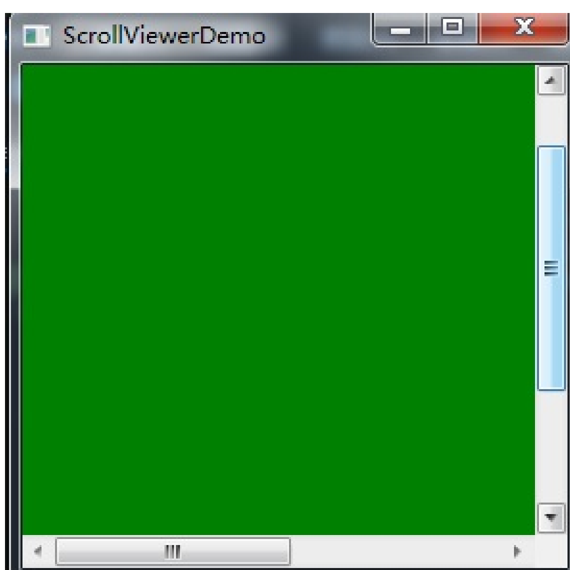


3.8 ScrollViewer 控件

通常用户界面中的内容比计算机屏幕的显示区域大的时候，可以利用ScrollViewer.aspx)控件可以方便地使应用程序中的内容具备滚动功能。具体的使用示例如下所示：

```
1 <Window x:Class="WPFLayoutDemo.ScrollViewerDemo"
2         xmlns="http://schemas.microsoft.com/winfx/2006/xaml/pres
3         xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4         Title="ScrollViewerDemo" Height="300" Width="300">
5     <Grid>
6         <ScrollViewer **HorizontalScrollBarVisibility="Visible"
7             <Rectangle Width="500" Height="400" Fill="Green"/>
8         </ScrollViewer>
9     </Grid>
10 </Window>
```

运行效果如下图所示：



四、布局综合运用

前面例子都是单独介绍每个布局控件的，然而在实际开发中，程序的界面布局都是由多个布局控件一起来完成的，这里演示一个综合实验的小例子。要实现的效果图如下所示：



具体的XAML代码实现如下所示：

```
1 <Window x:Class="WPFLayoutDemo.MainWindow"
2         xmlns="http://schemas.microsoft.com/winfx/2006/xaml/pres
3         xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4         WindowStartupLocation="CenterScreen"
5         Title="布局综合运用实例" Height="400" Width="480">
6     <DockPanel Width="Auto" Height="Auto" LastChildFill="True">
7         <!--顶部菜单区域-->
8         <Menu Width="Auto" Height="20" Background="LightGray" Do
9             <!--File菜单项-->
10            <MenuItem Header="文件">
11                <MenuItem Header="保存"/>
12                <Separator/>
13                <MenuItem Header="退出"/>
14            </MenuItem>
15            <!--About 菜单项-->
16            <MenuItem Header="帮助">
17                <MenuItem Header="关于本产品"/>
18            </MenuItem>
19        </Menu>
20
21        <!--状态栏-->
22        <StackPanel Width="Auto" Height="25" Background="LightGr
23            <Label Width="Auto" Height="Auto" Content="状态栏" Fo
24        </StackPanel>
25        <!--Left-->
26        <StackPanel Width="130" Height="Auto" Background="Gray"
27            <Button Margin="10" Width="Auto" Height="30" Content
28            <Button Margin="10" Width="Auto" Height="30" Content
29            <Button Margin="10" Width="Auto" Height="30" Content
```

```
30          </StackPanel>
31
32          <!--Right-->
33          <Grid Width="Auto" Height="Auto" Background="White">
34
35              <Grid.ColumnDefinitions>
36                  <ColumnDefinition Width="*"/>
37                  <ColumnDefinition Width="*"/>
38              </Grid.ColumnDefinitions>
39
40              <Grid.RowDefinitions>
41                  <RowDefinition Height="*"/>
42                  <RowDefinition Height="*"/>
43              </Grid.RowDefinitions>
44
45              <Rectangle Fill="Gray" Margin="10,10,10,10" Grid.Rov
46              <Rectangle Fill="Gray" Margin="10,10,10,10" Grid.Rov
47              <Rectangle Fill="Gray" Margin="10,10,10,10" Grid.Rov
48              <Rectangle Fill="Gray" Margin="10,10,10,10" Grid.Rov
49          </Grid>
50      </DockPanel>
51
52  </Window>
```

五、自定义布局控件

在实际开发中，自然少不了自定义控件的开发，下面介绍下如何自定义布局控件。在前面介绍过布局系统的工作原理是先测量后排列，测量即是确定面板需要多大空间，排列则是定义面板内子元素的排列规则。所以，要实现自定义布局控件，需要继承于Panel类并重写MeasureOverride和ArrangeOverride方法即可，下面实现了一个简单的自定义布局控件：

```
 1 namespace CustomLayoutControl
 2 {
 3     public class CustomStackPanel: Panel
 4     {
 5         public CustomStackPanel()
 6             : base()
 7         {
 8         }
 9
10         // 重写默认的Measure方法
11         // avaiableSize是自定义布局控件的可用大小
12         protected override Size MeasureOverride(Size availableSi
13         {
14             Size panelDesiredSize = new Size();
15             foreach (UIElement child in this.InternalChildren)
16             {
17                 child.Measure(availableSize);
18
19                 // 子元素的期望大小
20                 panelDesiredSize.Width += child.DesiredSize.Widt
21                 panelDesiredSize.Height += child.DesiredSize.Hei
22             }
23
24             return panelDesiredSize;
25         }
26
27         // 重写默认的Arrange方法
28         protected override Size ArrangeOverride(Size finalSize)
29         {
30             double x = 10;
31             double y = 10;
32             foreach (UIElement child in this.InternalChildren)
33             {
34                 // 排列子元素的位置
35                 child.Arrange(new Rect(new Point(x, y), new Size
36                 y += child.RenderSize.Height + 5;
37             }
38
39             return finalSize;
40         }
41     }
42 }
```

控件的最终大小和位置是由该控件和父控件共同完成的，父控件会先给子控件提供可用大小（MeasureOverride中availableSize参数），子控件再反馈给父控件一个自己的期望值（DesiredSize），父控件最后根据自己所拥有的空间大小与子控件期望的值分配一定的空间给子控件并返回自己的大小。这个过程是通过MeasureOverride和ArrangeOverride这两个方法共同完成的，这里需要注意：父控件的availableSize是减去Margin、Padding等的值。

接下来，创建一个测试上面自定义布局控件的WPF项目，然后添加自定义布局控件的程序集，然后在WPF项目中MainWindows添加如下代码：

```
 1 <Window x:Class="TestCustomerPanel.MainWindow"
 2         xmlns="http://schemas.microsoft.com/winfx/2006/xaml/pres
 3         xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 4 **      xmlns:custom="clr-namespace:CustomLayoutControl;assembly
 5         Title="测试自定义布局控件" Height="350" Width="525">
 6     <custom:CustomStackPanel Background="Red">
 7         <Button Content="Button 1"></Button>
 8         <Button Content="Button 2"></Button>
 9         <Button Content="Button 3"></Button>
10     </custom:CustomStackPanel>
11 </Window>
```

运行成功后的效果如下图所示：



六、小结

到这里，WPF布局的内容就介绍结束了，这里最后只是简单地定义了一个类似StackPanel的布局控件，你还可以自定义更加复杂的布局控件，关于更复杂的自定义控件，你可以参考如下一些文章。在下面一篇文章将分享WPF中依赖属性的内容。

- FishEyePanel/FanPanel
- PlotPanel, Windows SDK Sample

本文所有源码下载：WPFLayouDemo.zip

WPF快速入门系列(2)——深入解析依赖属性

一、引言

感觉最近都颓废了，好久没有学习写博文了，出于负罪感，今天强烈逼迫自己开始更新WPF系列。尽管最近看到一篇WPF技术是否老矣的文章，但是还是不能阻止我系统学习WPF。今天继续分享WPF中一个最重要的知识点——依赖属性。

二、依赖属性的全面解析

听到依赖属性，自然联想到C#中属性的概念。C#中属性是抽象模型的核心部分，而依赖属性是专门基于WPF创建的。在WPF库实现中，依赖属性使用普通的C#属性进行了包装，使得我们可以通过和以前一样的方式来使用依赖属性，但我们必须明确，在WPF中我们大多数都在使用依赖属性，而不是使用属性。依赖属性重要性在于，在WPF核心特性，如动画、数据绑定以及样式中都需要使用到依赖属性。既然WPF引入了依赖属性，也自然有其引入的道理。WPF中的依赖属性主要有以下三个优点：

- 依赖属性加入了属性变化通知、限制、验证等功能。这样可以使我们更方便地实现应用，同时大大减少了代码量。许多之前需要写很多代码才能实现的功能，在WPF中可以轻松实现。
- 节约内存：在WinForm中，每个UI控件的属性都赋予了初始值，这样每个相同的控件在内存中都会保存一份初始值。而WPF依赖属性很好地解决了这个问题，它内部实现使用哈希表存储机制，对多个相同控件的相同属性的值都只保存一份。关于依赖属性如何节约内存的更多内容参考：WPF的依赖属性是怎么节约内存的
- 支持多种提供对象：可以通过多种方式来设置依赖属性的值。可以配合表达式、样式和绑定来对依赖属性设置值。

2.1 依赖属性的定义

上面介绍了依赖属性所带来的好处，这时候，问题又来了，怎样自己定义一个依赖属性呢？C#属性的定义大家再熟悉不过了。下面通过把C#属性进行改写成依赖属性的方式来介绍依赖属性的定义。下面是一个属性的定义：

在把上面属性改写为依赖属性之前，下面总结下定义依赖属性的步骤：

1. 让依赖属性的所在类型继承自DependencyObject.aspx)类。
2. 使用public static 声明一个DependencyProperty.aspx)的变量，该变量就是真正的依赖属性。
3. 在类型的静态构造函数中通过Register.aspx)方法完成依赖属性的元数据注册。
4. 提供一个依赖属性的包装属性，通过这个属性来完成对依赖属性的读写操作。

根据上面的四个步骤，下面来把Name属性来改写成一个依赖属性，具体的实现代码如下所示：

```
// 1\. 使类型继承DependencyObject类
    public class Person : DependencyObject
    {
        // 2\. 声明一个静态只读的DependencyProperty 字段
        public static readonly DependencyProperty nameProperty;

        static Person()
        {
            // 3\. 注册定义的依赖属性
            nameProperty = DependencyProperty.Register("Name", type
                new PropertyMetadata("Learning Hard",OnValueChanged
        }

        // 4\. 属性包装器，通过它来读取和设置我们刚才注册的依赖属性
        public string Name
        {
            get { return (string)GetValue(nameProperty); }
            set { SetValue(nameProperty, value); }
        }

        private static void OnValueChanged(DependencyObject dpobj,
        {
            // 当只发生改变时回调的方法
        }

    }
```

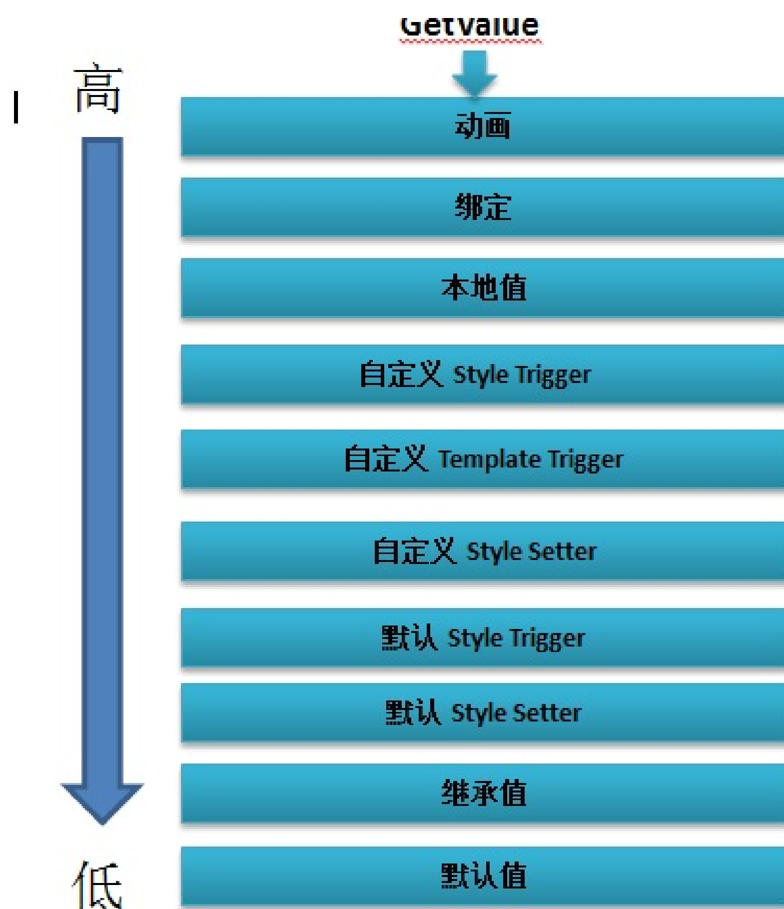
从上面代码可以看出，依赖属性是通过调用**DependencyObject**的**GetValue**和**SetValue**来对依赖属性进行读写的。它使用哈希表来进行存储的，对应的Key就是属性的HashCode值，而值（Value）则是注册的DependencyPropery；而C#中的属性是类私有字段的封装，可以通过对该字段进行操作来对属性进行读写。总结为：属性是字段的包装，**WPF**中使用属性对依赖属性进行包装。

2.2 依赖属性的优先级

WPF允许在多个地方设置依赖属性的值，则自然就涉及到依赖属性获取值的优先级问题。例如下面XMAL代码，我们在三个地方设置了按钮的背景颜色，那最终按钮会读取那个设置的值呢？是Green、Yellow还是Red？

```
<Window x:Class="DPSample.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/present
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Button x:Name="myButton" **Background="Green"** Width="10(
            <Button.Style>
                <Style TargetType="{x:Type Button}">
                    <Setter **Property="Background" Value="Yellow"
                    <Style.Triggers>
                        <Trigger Property="IsMouseOver" Value="True
                            <Setter **Property="Background" Value="
                        </Trigger>
                    </Style.Triggers>
                </Style>
            </Button.Style>
            Click Me
        </Button>
    </Grid>
</Window>
```

上面按钮的背景颜色是Green。之所以背景色是Green，是因为WPF每访问一个依赖属性，它都会按照下面的顺序由高到底处理该值。具体优先级如下图所示：



在上面XAML中，按钮的本地值设置的是Green，自定义Style Trigger设置的为Red，自定义的Style Setter设置的为Yellow，由于这里的本地值的优先级最高，所以按钮的背景色或者的是Green值。如果此时把本地值Green去掉的话，此时按钮的背景颜色是Yellow而不是Red。这里尽管Style Trigger的优先级比Style Setter高，但是由于此时Style Trigger的IsMouseOver属性为false，即鼠标没有移到按钮上，一旦鼠标移到按钮上时，此时按钮的颜色就为Red。此时才会体现出Style Trigger的优先级比Style Setter优先级高。所以上图中优先级是比较理想情况下，很多时候还需要具体分析。

2.3 依赖属性的继承

依赖属性是可以被继承的，即父元素的相关设置会自动传递给所有的子元素。下面代码演示了依赖属性的继承。

```
<Window x:Class="Custom_DPInherited.DPInherited"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentat
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      xmlns:mc="http://schemas.openxmlformats.org/markup-compatibil
      xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
      mc:Ignorable="d"
      d:DesignHeight="300" d:DesignWidth="300"
      FontSize="18"
      Title="依赖属性的继承">
    <StackPanel >
        <Label Content="继承自Window的FontSize" />
        <Label Content="显式设置FontSize"
               TextElement.FontSize="36"/>
        <StatusBar>Statusbar没有继承自Window的FontSize</StatusBar>
    </StackPanel>
</Window>
```

上面的代码的运行效果如下图所示：



在上面XAML代码中。Window.FontSize设置会影响所有内部子元素字体大小，这就是依赖属性的继承。如第一个Label没有定义FontSize，所以它继承了Window.FontSize值。但一旦子元素提供了显式设置，这种继承就会被打断，所以Window.FontSize值对于第二个Label不再起作用。

这时，你可能已经发现了问题：StatusBar没有显式设置FontSize值，但它的字体大小没有继承Window.FontSize的值，而是保持了系统的默认值。那这是什么原因呢？其实导致这样的问题：并不是所有元素都支持属性值继承的，如StatusBar、Tooptip和Menu控件。另外，StatusBar等控件截获了从父元素继承来的属性，并且该属性也不会影响StatusBar控件的子元素。例如，如果我们在StatusBar中添加一个Button。那么这个Button的FontSize属性也不会发生改变，其值为默认值。

前面介绍了依赖属性的继承，那我们如何把自定义的依赖属性设置为可被其他控件继承呢？通过AddOwer.aspx)方法可以依赖属性的继承。具体的实现代码如下所示：

```
1  public class CustomStackPanel : StackPanel
2      {
3          public static readonly DependencyProperty MinDatePropert
4
5          static CustomStackPanel()
6          {
7              MinDateProperty = DependencyProperty.Register("MinDa
8          }
9
10         public DateTime MinDate
11         {
12             get { return (DateTime)GetValue(MinDateProperty); }
13             set { SetValue(MinDateProperty, value); }
14         }
15     }
16
17     public class CustomButton :Button
18     {
19         private static readonly DependencyProperty MinDatePrope
20
21         static CustomButton()
22         {
23             // AddOwner方法指定依赖属性的所有者，从而实现依赖属性的继承，
24             // 注意FrameworkPropertyMetadataOptions的值为Inherits
25             MinDateProperty = CustomStackPanel.MinDateProperty.A
26         }
27
28         public DateTime MinDate
29         {
30             get { return (DateTime)GetValue(MinDateProperty); }
31             set { SetValue(MinDateProperty, value); }
32         }
33     }
```

接下来，你可以在XAML中进行测试使用，具体的XAML代码如下：

```
<Window x:Class="Custom_DPInherited.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/present
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:Custom_DPInherited"
        xmlns:sys="clr-namespace:System;assembly=mscorlib"
        Title="实现自定义依赖属性的继承" Height="350" Width="525">
    <Grid>
        <local:CustomStackPanel x:Name="customStackPanle" MinDate='
            <!--CustomStackPanel的依赖属性-->
            <ContentPresenter Content="{Binding Path=MinDate, Eleme
            <local:CustomButton Content="{Binding RelativeSource={x
        </local:CustomStackPanel>
    </Grid>
</Window>
```

上面XAML代码中，显示设置了CustomStackPanel的MinDate的值，而在CustomButton中却没有显式设置其MinDate值。CustomButton的Content属性的值是通过绑定MinDate属性来进行获取的，关于绑定的更多内容会在后面文章中分享。在这里CustomButton中并没有设置MinDate的值，但是CustomButton的Content的值却是当前的时间，从而可以看出，此时CustomButton的MinDate属性继承了CustomStackPanel的MinDate的值，从而设置了其Content属性。最终的效果如下图所示：



2.4 只读依赖属性

在C#属性中，我们可以通过设置只读属性来防止外界恶意更改该属性值，同样，在WPF中也可以设置只读依赖属性。如IsMouseOver.aspx)就是一个只读依赖属性。那我们如何创建一个只读依赖属性呢？其实只读的依赖属性的定义方式与一般依赖属性的定义方式基本一样。只读依赖属性仅仅是用DependencyProperty.RegisterReadonly.aspx)替换了DependencyProperty.Register而已。下面代码实现了一个只读依赖属性。

```
 1 public partial class MainWindow : Window
 2     {
 3         public MainWindow()
 4         {
 5             InitializeComponent();
 6 
 7             // 内部使用SetValue来设置值
 8             SetValue(counterKey, 8);
 9         }
10 
11         // 属性包装器，只提供GetValue，你也可以设置一个private的SetVal
12         public int Counter
13         {
14             get { return (int)GetValue(counterKey.DependencyProp
15         }
16 
17         // 使用RegisterReadOnly来代替Register来注册一个只读的依赖属性
18         private static readonly DependencyPropertyKey counterKey
19             DependencyProperty.RegisterReadOnly("Counter",
20             typeof(int),
21             typeof(MainWindow),
22             new PropertyMetadata(0));
23     }
```

对应的XAML代码为：

```
<Window x:Class="ReadOnlyDP.MainWindow"
        Name="ThisWin"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/present
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ReadOnly Dependency Property" Height="350" Width="52
    <Grid>
        <Viewbox>
            <TextBlock Text="{Binding ElementName=ThisWin, Path=Cou
        </Viewbox>
    </Grid>
</Window>
```

此时Counter包装的counterKey就是一个只读依赖属性，因为其定义为private的，所以在类外也不能使用DependencyObject.SetValue方法来对其值，而包装的Counter属性又只提供了GetValue方法，所以类外部只能对该依赖属性进行读取，而不能对其赋值。此时运行效果如下图所示。



2.5 附加属性

WPF中还有一类特殊的属性——附加属性。附加是一种特殊的依赖属性。它允许给一个对象添加一个值，而该对象可能对这个值一无所知。附加属性最常见的例子就是布局容器中DockPanel类中的Dock附加属性和Grid类中Row和Column附加属性。那问题又来了，我们怎样在自己的类中定义一个附加属性呢？其实定义附加属性和定义一般的依赖属性一样没什么区别，只是用RegisterAttached方法代替了Register方法罢了。下面代码演示了附加属性的定义。

```
public class AttachedPropertyClass
    {
        // 通过使用RegisterAttached来注册一个附加属性
        public static readonly DependencyProperty IsAttachedPropert
            DependencyProperty.RegisterAttached("IsAttached", typeo
            new FrameworkPropertyMetadata((bool)false));

        // 通过静态方法的形式暴露读的操作
        public static bool GetIsAttached(DependencyObject dpo)
        {
            return (bool)dpo.GetValue(IsAttachedProperty);
        }

        public static void SetIsAttached(DependencyObject dpo, bool
        {
            dpo.SetValue(IsAttachedProperty, value);
        }
    }
```

在上面代码中，IsAttached就是一个附加属性，附加属性没有采用CLR属性进行封装，而是使用静态SetIsAttached方法和GetIsAttached方法来存取IsAttached值。这两个静态方法内部一样是调用SetValue和GetValue来对附加属性读写的。

2.6 依赖属性验证和强制

在定义任何类型的属性时，都需要考虑错误设置属性的可能性。对于传统的CLR属性，可以在属性的设置器中进行属性值的验证，不满足条件的值可以抛出异常。但对于依赖属性来说，这种方法不合适，因为依赖属性通过SetValue方法来直接设置其值的。然而WPF有其代替的方式，WPF中提供了两种方法来用于验证依赖属性的值。

- ValidateValueCallback:该回调函数可以接受或拒绝新值。该值可作为DependencyProperty.Register.aspx)方法的一个参数。
- CoerceValueCallback:该回调函数可将新值强制修改为可被接受的值。例如某个依赖属性Age的值范围是0到120，在该回调函数中，可以对设置的值进行强制修改，对于不满足条件的值，强制修改为满足条件的值。如当设置为负值时，可强制修改为0。该回调函数可作为PropertyMetadata.aspx)构造函数参数进行传递。

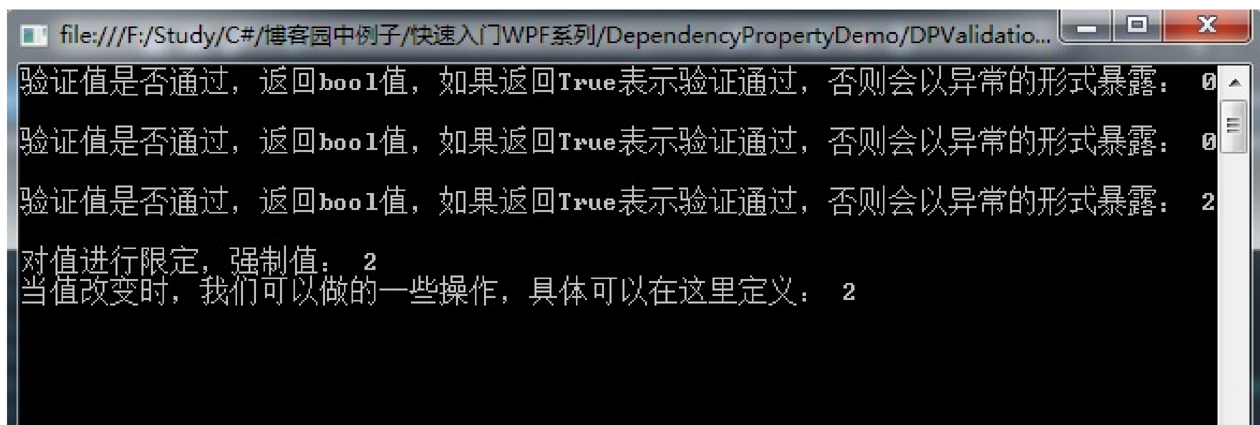
当应用程序设置一个依赖属性时，所涉及的验证过程如下所示：

1. 首先，CoerceValueCallback方法可以修改提供的值或返回DependencyProperty.UnsetValue.aspx)。
2. 如果CoerceValueCallback方法强制修改了提供的值，此时会激活ValidateValueCallback方法进行验证，如果该方法返回为true，表示该值合法，被认为可被接受的，否则拒绝该值。不像CoerceValueCallback方法，ValidateValueCallback方法不能访问设置属性的实际对象，这意味着你不能检查其他属性值。即该方法中不能对类的其他属性值进行访问。
3. 如果上面两个阶段都成功的话，最后会触发PropertyChangedCallback方法来触发依赖属性值的更改。

下面代码演示了基本的流程。

```
 1 class Program
 2     {
 3         static void Main(string[] args)
 4         {
 5             SimpleDPClass sDPClass = new SimpleDPClass();
 6             sDPClass.SimpleDP = 2;
 7             Console.ReadLine();
 8         }
 9     }
10
11     public class SimpleDPClass : DependencyObject
12     {
13         public static readonly DependencyProperty SimpleDPProper
14             DependencyProperty.Register("SimpleDP", typeof(doubl
15                 new FrameworkPropertyMetadata((double)0.0,
16                     FrameworkPropertyMetadataOptions.None,
17                     new PropertyChangedCallback(OnValueChanged),
18                     new CoerceValueCallback(CoerceValue)),
19                     new ValidateValueCallback(IsValidValue));
20
21         public double SimpleDP
22         {
23             get { return (double)GetValue(SimpleDPProperty); }
24             set { SetValue(SimpleDPProperty, value); }
25         }
26
27         private static void OnValueChanged(DependencyObject d, D
28         {
29             Console.WriteLine("当值改变时，我们可以做的一些操作，具体可
30         }
31
32         private static object CoerceValue(DependencyObject d, ob
33         {
34             Console.WriteLine("对值进行限定，强制值： {0}", value);
35             return value;
36         }
37
38         private static bool IsValidValue(object value)
39         {
40             Console.WriteLine("验证值是否通过，返回bool值，如果返回Tr
41             return true;
42         }
43     }
```

其运行结果如下图所示：



从运行结果可以看出，此时并没有按照上面的流程先Coerce后Validate的顺序执行，这可能是WPF内部做了一些特殊的处理。当属性被改变时，首先会调用Validate来判断传入的value是否有效，如果无效就不继续后续操作。并且CoerceValue后面并没有运行ValidateValue，而是直接调用PropertyChanged。这是因为CoerceValue操作并没有强制改变属性的值，而前面对这个值已经验证过了，所以也就没有必要再运行Valudate方法来进行验证了。但是如果在Coerce中改变了Value的值，那么还会再次调用Valudate操作来验证值是否合法。

2.7 依赖属性的监听

我们可以用两种方法对依赖属性的改变进行监听。这两种方法是：

- 使用DependencyPropertyDescriptor.aspx)类
- 使用OverrideMetadata.aspx)的方式。

下面分别使用这两种方式来实现下对依赖属性的监听。

第一种方式：定义一个派生于依赖属性所在的类，然后重写依赖属性的元数据并传递一个PropertyChangedCallback参数即可，具体的实现如下代码所示：

```
1 public class MyTextBox : TextBox
2     {
3         public MyTextBox()
4             : base()
5         {
6         }
7 
8         static MyTextBox()
9         {
10            //第一种方法，通过OverrideMetadata
11            TextProperty.OverrideMetadata(typeof(MyTextBox), new
12        }
13
14        private static void TextPropertyChanged(DependencyObject
15        {
16            MessageBox.Show("", "Changed");
17        }
18    }
```

第二种方法：这个方法更加简单，获取DependencyPropertyDescriptor并调用AddValueChange方法为其绑定一个回调函数。具体实现代码如下所示：

```
public MainWindow()
       {
           InitializeComponent();
           //第二种方法，通过OverrideMetadata
           DependencyPropertyDescriptor descriptor = DependencyPro
           descriptor.AddValueChanged(tbxEditMe, tbxEditMe_TextCha
       }

       private void tbxEditMe_TextChanged(object sender, EventArgs
       {
           MessageBox.Show("", "Changed");
       }
```

三、总结

到这里，依赖属性的介绍就结束了。WPF中的依赖属性通过一个静态只读字段进行定义，并且在静态构造函数中进行注册，最后通过.NET传统属性进行包装，使其使用与传统的.NET属性并无两样。在后面一篇文章将分享WPF中新的事件机制——路由事件。

本文所有源码下载：DependencyPropertyDemo.zip

WPF快速入门系列(3)——深入解析WPF事件机制

一、引言

WPF除了创建了一个新的依赖属性系统之外，还用更高级的路由事件功能替换了普通的.NET事件。

路由事件是具有更强传播能力的事件——它可以在元素树上向上冒泡和向下隧道传播，并且沿着传播路径被事件处理程序处理。与依赖属性一样，可以使用传统的事件方式使用路由事件。尽管路由事件的使用方式与传统的事件一样，但是理解其工作原理还是相当重要的。

二、路由事件的详细介绍

对于.NET中的事件，大家应该在熟悉不过了。事件指的在某个事情发生时，由对象发送用于通知代码的消息。WPF中的路由事件允许事件可以被传递。例如，路由事件允许一个来自工具栏按钮的单击事件，在被处理之前可以传递到工具栏，然后再传递到包含工具栏的窗口。那么现在问题来了，我怎样在WPF中去定义一个路由事件呢？

2.1 如何定义路由事件

既然有了问题，自然就要去解决了。在自己定义一个依赖属性之前，首先，我们得学习下WPF框架中是怎么去定义的，然后按照WPF框架中定义的方式去试着自己定义一个依赖属性。下面通过Reflector工具来查看下WPF中Button.aspx)按钮的Click事件的定义方式。

由于Button按钮的Click事件是继承于ButtonBase.aspx)基类的，所以我们直接来查看ButtonBase中Click事件的定义。具体的定义代码如下所示：

```
[Localizability(LocalizationCategory.Button), DefaultEvent("Click")
public abstract class ButtonBase : ContentControl, ICommandSource
{
    // 事件定义
    **public static readonly RoutedEvent ClickEvent;** // 事件注册
    static ButtonBase()
    {
       **ClickEvent** **= EventManager.RegisterRoutedEvent("Click",
        CommandProperty = DependencyProperty.Register("Command", ty
      .......
    }

    // 传统事件包装
   ** public event RoutedEventHandler Click
    {
        add
        {
            base.AddHandler(ClickEvent, value);
        }
        remove
        {
            base****.RemoveHandler(ClickEvent, value);
        }
    }**
    .......
}
```

从上面代码可知，路由事件的定义与依赖属性的定义类似，路由事件由只读的静态字段表示，在一个静态构造函数通过EventManager.RegisterRoutedEvent.aspx)函数注册，并且通过一个.NET事件定义进行包装。

现在已经知道了路由事件是如何在WPF框架中定义和实现的了，那要想自己定义一个路由事件也自然不在话下了。

2.2 共享路由事件

与依赖属性一样，可以在类之间共享路由事件的定义。即实现路由事件的继承。例如UIElement.aspx)类和ContentElement类都使用了MouseUp事件，但MouseUp事件是由System.Windows.Input.Mouse.aspx)类定义的。UIElement类和ContentElement类只是通过RouteEvent.AddOwner.aspx)方法重用了MouseUp事件。你可以在UIElement类的静态构造函数找到下面的代码：

```
static UIElement()
{
    _typeofThis = typeof(UIElement);

     PreviewMouseUpEvent =  Mouse.PreviewMouseUpEvent.AddOwner(_typ
    MouseUpEvent = **Mouse.MouseUpEvent.AddOwner(_typeofThis)**;
}
```

2.3 引发和处理路由事件

尽管路由事件通过传统的.NET事件进行包装，但路由事件并不是通过.NET事件触发的，而是使用RaiseEvent方法触发事件，所有元素都从UIElement类继承了该方法。下面代码是具体ButtonBase类中触发路由事件的代码：

而在WinForm中，Button.aspx)的Click事件是通过调用委托进行触发的，具体的实现代码如下所示：

```
1  protected virtual void OnClick(EventArgs e)
2         {
3             EventHandler handler = (EventHandler)base.Events[Ever
4             if (handler != null)
5             {
6                 **handler(this, e);** // 直接调用委托进行触发事件
7             }
8         }
```

对于路由事件的处理，与原来WinForm方式一样，你可以在XAML中直接连接一个事件处理程序，具体实现代码如下所示：

```
<TextBlock Margin="3" MouseUp="SomethingClick" Name="tbxTest">
    text label
</TextBlock>
// 后台cs代码
private void SomethingClick(object sender, MouseButtonEventArgs e)
{
}
```

同时还可以通过后台代码的方式连接事件处理程序，具体的实现代码如下所示：

三、路由事件其特殊性

路由事件的特殊性在于其传递性，WPF中的路由事件分为三种。

- 与普通的.NET事件类似的直接路由事件(Direct event)。它源自一个元素，并且不传递给其他元素。例如，MouseEnter事件(当鼠标移动到一个元素上面时触发)就是一个直接路由事件。
- 在包含层次中向上传递的冒泡路由事件(**Bubbling event)**。例如，MouseDown事件就是一个冒泡路由事件。它首先被单击的元素触发，接下来就是该元素的父元素触发，依此类推，直到WPF到达元素树的顶部为止。
- 在包含层次中向下传递的隧道路由事件(**Tunneling event)**。例如PreviewKeyDown就是一个隧道路由事件。在一个窗口上按下某个键，首先是窗口，然后是更具体的容器，直到到达按下键时具有焦点的元素。

既然，路由事件有三种表现形式，那我们怎么去区别具体的路由事件是属于哪种呢？辨别的方法在于路由事件的注册方法上，当使用EventManager.RegisterEvent.aspx)方法注册一个路由事件时，需要传递一个RoutingStrategy.aspx)枚举值来标识希望应用于事件的事件行为。

3.1 冒泡路由事件

下面代码演示了事件冒泡过程：

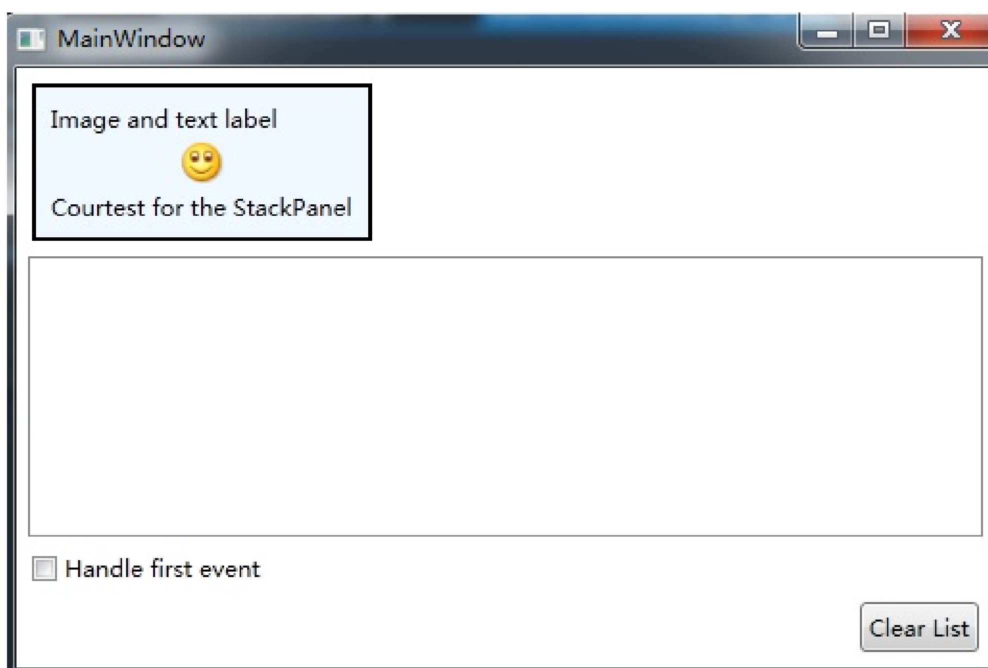
```
<Window x:Class="BubbleLabelClick.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/present
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525" MouseUp="Someth
    <Grid Margin="3" MouseUp="SomethingClick">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>
        <Label Margin="5" Grid.Row="0" HorizontalAlignment="Left" B
               BorderBrush="Black" BorderThickness="2" MouseUp="Som
            <StackPanel MouseUp="SomethingClick">
                <TextBlock Margin="3" MouseUp="SomethingClick" Name
                    Image and text label
                </TextBlock>
                <Image Source="pack://application:,,,/BubbleLabelCl
                <TextBlock Margin="3" MouseUp="SomethingClick">
                    Courtest for the StackPanel
                </TextBlock>
            </StackPanel>
        </Label>

        <ListBox Grid.Row="1" Margin="3" Name="lstMessage">
        </ListBox>
        <CheckBox Margin="5" Grid.Row="2" Name="chkHandle">Handle f
        <Button Click="cmdClear_Click"  Grid.Row="3" HorizontalAlig
    </Grid>
</Window>
```

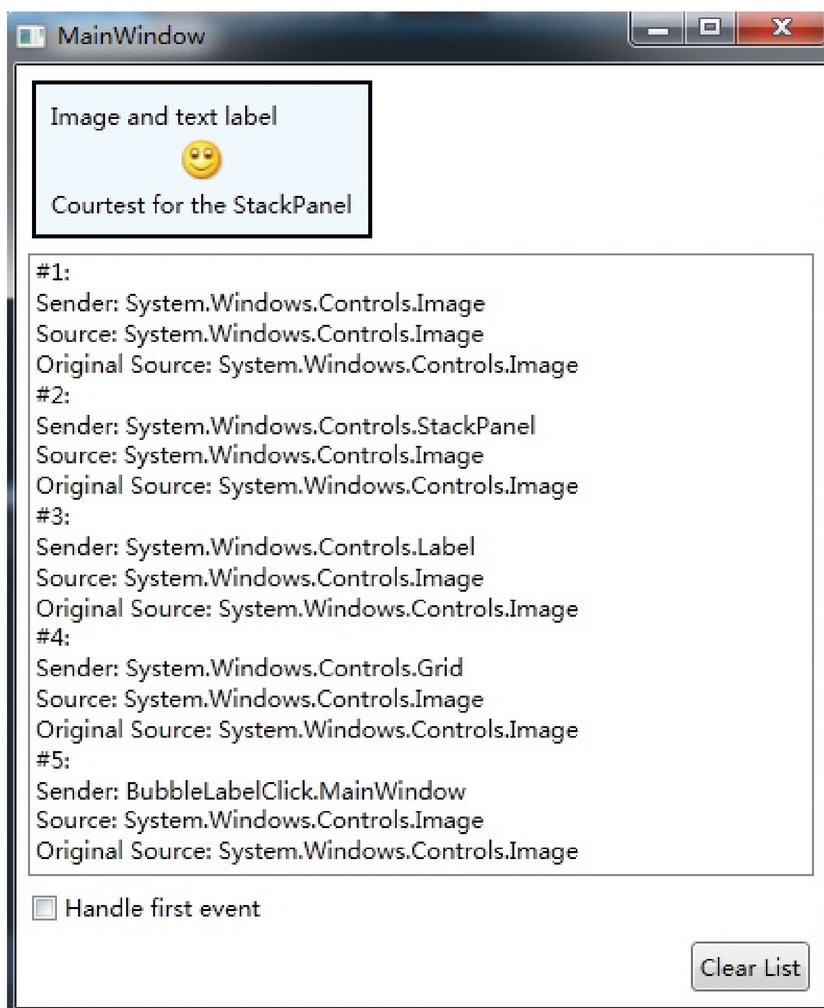
其后台代码为：

```
1      public partial class MainWindow : Window
2      {
3          public MainWindow()
4          {
5              InitializeComponent();
6
7          }
8
9          private int eventCounter = 0;
10
11         private void SomethingClick(object sender, RoutedEventAr
12         {
13             eventCounter++;
14             string message = "#" + eventCounter.ToString() + ":\
15                 "Source: " + e.Source + "\r\n" +
16                 "Original Source: " + e.OriginalSource;
17             lstMessage.Items.Add(message);
18             e.Handled = (bool)chkHandle.IsChecked;
19         }
20
21         private void cmdClear_Click(object sender, RoutedEventAr
22         {
23             eventCounter = 0;
24             lstMessage.Items.Clear();
25         }
26     }
```

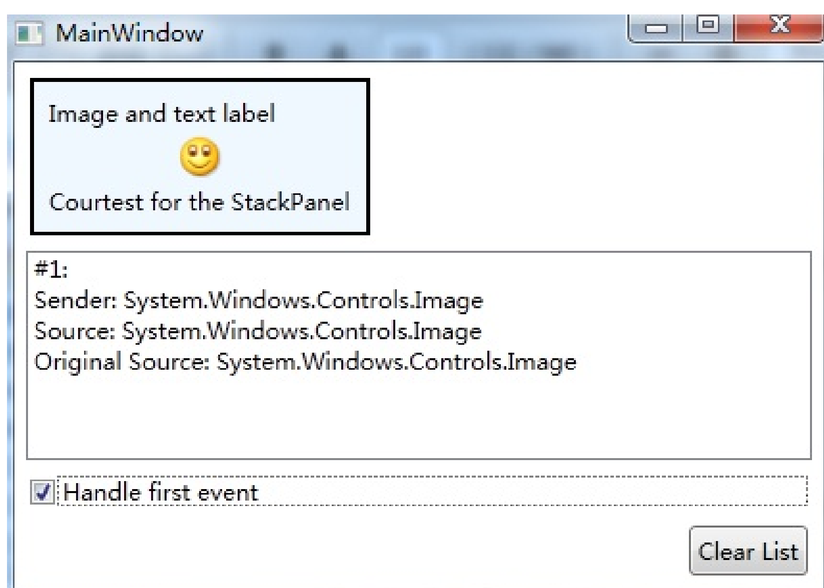
运行之后的效果图如下所示：



单击窗口中的笑脸图像之后，程序的运行结果如下图所示。



从上图结果可以发现，MouseUp事件由下向上传递了5级，直到窗口级别结束。另外，如果选择了Handle first event复选框的话，SomethingClicked方法会将RoutedEventArgs.Handled属性设置为true，表示事件已被处理，且该事件将终止向上冒泡。因此，此时列表中只能看到Image的事件，具体运行结果如下图所示：



并且在列表框或窗口空白处进行单击，此时也一样只会出现一次MouseUp事件。但单击一个地方例外。当单击Clear List按钮，此时不会引发MouseUp事件。这是因为按钮包含一些特殊的处理代码，这些代码会挂起MouseUp事件（即不会触发MouseUp事件，则相应的事件处理程序也不会被调用），并引发一个更高级的Click事件，同时，Handled标记被设置为true(这里指的在触发Click事件时会把Handled设置为true)，从而阻止MouseUp事件继续向上传递。

通过博友**yiifans**指出，上面有一点说错了，在设置**Handled = true**的时候，不管是冒泡还是隧道事件，它还是会继续传播的，只是对应的事件不会再处理了。这里之所以没有删除上面错误解释而是在这里另行说明，是为了**强**调，因为**WPF**编程宝典上也是说会阻止传播。如果想继续响应相应事件的话，可以通过**AddHandler.aspx)**方法进行注册。此时你可以去掉**XAMLStackPanel**中**MouseUp**的注册，而是通过后台代码的方式进行注册**MouseUp**事件，具体的实现代码如下：

之所以还是会继续上传，其实通过在SomethingClick事件处理程序中设置一个断点就可以发现其调用堆栈，具体的堆栈调用截图如下所示：

调用堆栈

名称	语言
BubbleLabelClick.exe!BubbleLabelClick.MainWindow.SomethingClick(object sender, System.Windows.Rout	C#
PresentationCore.dll!System.Windows.Input.MouseButtonEventArgs.InvokeEventHandler(System.Delegate	
PresentationCore.dll!System.Windows.RoutedEventArgs.InvokeHandler(System.Delegate handler, object tar	
PresentationCore.dll!System.Windows.RoutedEventHandlerInfo.InvokeHandler(object target, System.Windo	
PresentationCore.dll!System.Windows.EventRoute.InvokeHandlersImpl(object source, System.Windows.Rou	
PresentationCore.dll!System.Windows.UIElement.RaiseEventImpl(System.Windows.DependencyObject senc	
PresentationCore.dll!System.Windows.UIElement.RaiseTrustedEvent(System.Windows.RoutedEventArgs arg	
PresentationCore.dll!System.Windows.UIElement.RaiseEvent(System.Windows.RoutedEventArgs args, bool	
PresentationCore.dll!System.Windows.Input.InputManager.ProcessStagingArea() + 0x1f8 字节	
PresentationCore.dll!System.Windows.Input.InputManager.ProcessInput(System.Windows.Input.InputEvent	
PresentationCore.dll!System.Windows.Input.InputProviderSite.ReportInput(System.Windows.Input.InputRe	
PresentationCore.dll!System.Windows.Interop.HwndMouseInputProvider.ReportInput(System.IntPtr hwnd,	
PresentationCore.dll!System.Windows.Interop.HwndMouseInputProvider.FilterMessage(System.IntPtr hwnc	
PresentationCore.dll!System.Windows.Interop.HwndSource.InputFilterMessage(System.IntPtr hwnd, int ms	
WindowsBase.dll!MS.Win32.HwndWrapper.WndProc(System.IntPtr hwnd, int msg, System.IntPtr wParam, S	

从上图可以知道SomethingClick调用前都执行了哪些操作，其中RoutedEventHandleInfo.InvokeHandler方法的实现代码就是这个问题的关键所在，下面通过Reflector查看下这个方法的源码，具体查看的源码如下所示：

```
internal void InvokeHandler(object target, RoutedEventArgs routedE
{
    **if (!routedEventArgs.Handled || this****._handledEventsToo)**
    {
        if (this._handler is RoutedEventHandler)
        {
            ((RoutedEventHandler) this._handler)(target, routedEve
        }
        else
        {
            routedEventArgs.InvokeHandler(this._handler, target);
        }
    }
}
```

在上面代码中，红色标记的就是解释这个问题的关键代码，每当触发事件处理程序之前，都会检查**RoutedEventArgs**的**Handled**属性和**handleEventsToo**字段。这样我们就彻底明白了，当**Handle=true**时，其实路由事件一样还是会传递，只是传递到对应事件处理程序中时，只是因为**Handle**为**true**和**_handleEventsToo**为**false**，从而导致事件处理程序没有运行罢了，如果通过**AddHandler**（**RoutedEvent, Delegate, Boolean**）注册事件处理程序的话，此时把**_handleEventToo**显式设置为**true**了，所以即使**Handle**为**true**，该元素的事件处理程序照样会执行，因为此时**if**条件一样为**true**的。

3.2 隧道路由事件

隧道路由事件与冒泡路由事件的工作方式一样，只是方向相反。即如果上面的例子中，触发的是一个隧道路由事件的话，如果在图像上单击，则首先窗口触发该隧道路由事件，然后才是Grid控件，接下来是StackPanel面板，以此类推，直到到达实际源头，即标签中的图像为止。

看了上面的介绍。隧道路由事件想必是相当好理解吧。它与冒泡路由事件的传递方式相反。但是我们怎样去区别隧道路由事件呢？隧道路由事件的识别相当容易，因为隧道路由事件都是以单词**Preview**开头。并且，WPF一般都成对地定义冒泡路由事件和隧道路由事件。这意味着如果发现一个冒泡的MouseUp事件，则对应的PreviewMouseUp就是一个隧道路由事件。另外，隧道路由事件总是在冒泡路由事件之前被触发。

另外需要注意的一点是：如果将隧道路由事件标记为已处理的，那么冒泡路由事件就不会发生。这是因为这两个事件共享同一个RoutedEventArgs类的实例。隧道路由事件对于来执行一些预处理操作非常有用，例如，根据键盘上特定的键执行特定操作，或过滤掉特定的鼠标操作等这样的场景都可以在隧道路由事件处理程序中进行处理。下面的示例演示了PreviewKeyDown事件的隧道过程。XAML代码如下所示。

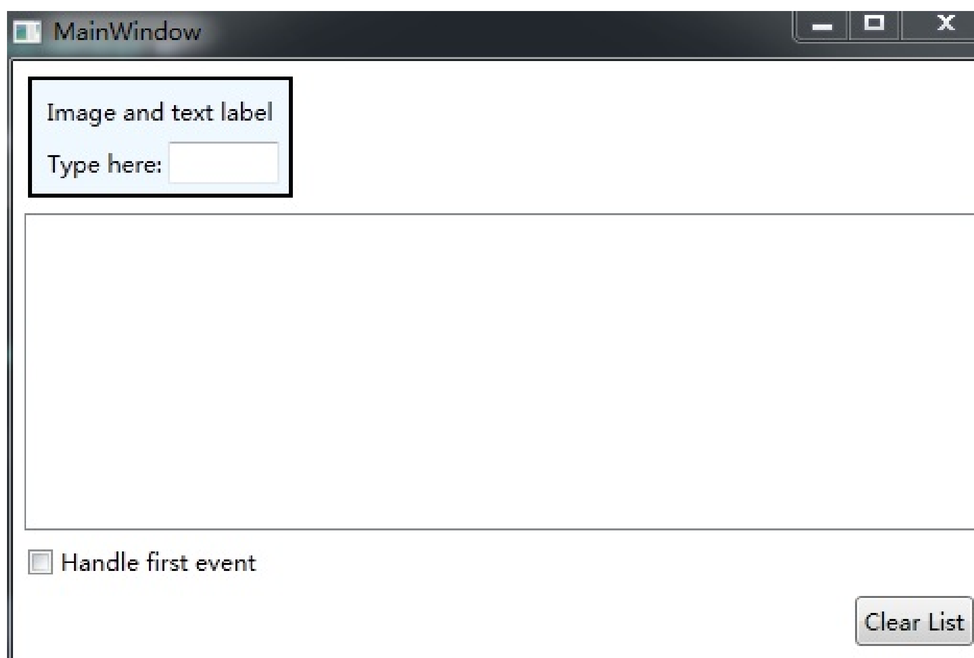
```
<Window x:Class="TunneleEvent.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/present
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525" PreviewKeyDown=
    <Grid Margin="3" PreviewKeyDown="SomeKeyPressed">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>
        <Label Margin="5" Grid.Row="0" HorizontalAlignment="Left" B
               BorderBrush="Black" BorderThickness="2" PreviewKeyDo
            <StackPanel>
                <TextBlock Margin="3" PreviewKeyDown="SomeKeyPresse
                    Image and text label
                </TextBlock>
                <Image Source="face.png" Stretch="None" PreviewMous
                <DockPanel Margin="0,5,0,0"  PreviewKeyDown="SomeKe
                    <TextBlock Margin="3"
                     PreviewKeyDown="SomeKeyPressed">
                        Type here:
                    </TextBlock>
                    <TextBox PreviewKeyDown="SomeKeyPressed" KeyDow
                </DockPanel>
            </StackPanel>
        </Label>

        <ListBox Grid.Row="1" Margin="3" Name="lstMessage">
        </ListBox>
        <CheckBox Margin="5" Grid.Row="2" Name="chkHandle">Handle t
        <Button Click="cmdClear_Click"  Grid.Row="3" HorizontalAlig
    </Grid>
</Window>
```

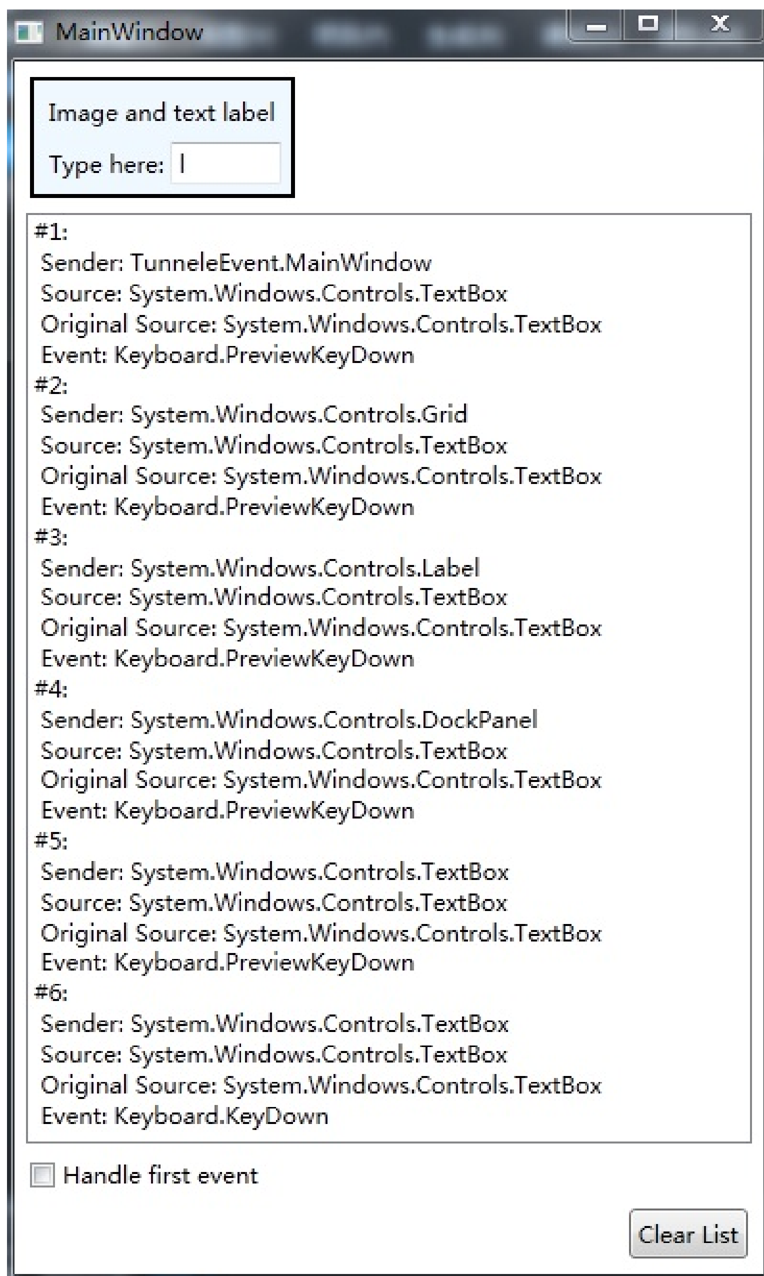
其对应的后台cs代码实现如下所示：

```
1 public partial class MainWindow : Window
2     {
3         public MainWindow()
4         {
5             InitializeComponent();
6         }
7
8         private int eventCounter = 0;
9
10        private void SomeKeyPressed(object sender, RoutedEventAr
11        {
12            eventCounter++;
13            string message = "#" + eventCounter.ToString() + ":\
14                " Sender: " + sender.ToString() + "\r\n" +
15                " Source: " + e.Source + "\r\n" +
16                " Original Source: " + e.OriginalSource + "\r\n'
17                " Event: " + e.RoutedEvent;
18            lstMessage.Items.Add(message);
19            e.Handled = (bool)chkHandle.IsChecked;
20        }
21
22        private void cmdClear_Click(object sender, RoutedEventAr
23        {
24            eventCounter = 0;
25            lstMessage.Items.Clear();
26        }
27    }
```

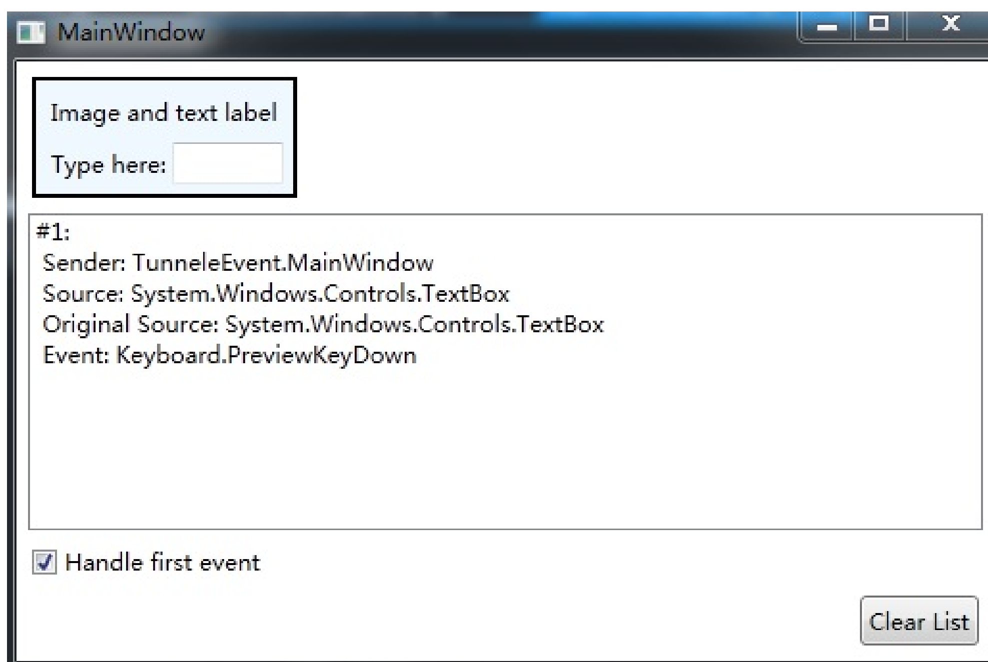
程序运行后的效果图如下所示：



在文本框中按下一个键时，事件首先在窗口触发，然后在整个层次结构中向下传递。具体的运行结果如下图所示：



如果在任何位置将**PreviewKeyDown**事件标记为已处理，则冒泡的**KeyDown**事件也就不会触发。当勾选了Handle first event 复选框时，当在输入框中按下一个键时，listbox中显示的记录只有1条记录，因为窗口触发的PrevieKeyDown事件处理已经把隧道路由事件标识为已处理，所以PreviewKeyDown事件将不会向下传递，所以此时只会显示一条MainWindow触发的记录。并且，此时，你可以注意到，我们按下的键上对应的字符并没有在输入框中显示，因为此时并没有触发**Textbox**中的**KeyDown**事件，因为改变文本框内容的处理是在**KeyDown**事件中处理的。具体的运行结果如下图所示：



3.3 附加事件

在上面例子中，因为所有元素都支持MouseUp和PreviewKeyDown事件。然而，许多控件都有它们自己特殊的事件。例如按钮的的Click事件，其他任何类都有定义该事件。假设有这样一个场景，StackPanel面板中包含了一堆按钮，并且希望在一个事件处理程序中处理所有这些按钮的单击事件。首先想到的办法就是将每个按钮的Click事件关联到同一个事件处理程序。但是Click事件支持事件冒泡，从而有一种更好的解决办法。可以在更高层次元素来关联Click事件来处理所有按钮的单击事件，具体的XAML代码实现如下所示：

```
<Window x:Class="AttachClickEvent.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/present
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <StackPanel Margin="3" **Button.Click="DoSomething"**>
        <Button Name="btn1">Button 1</Button>
        <Button Name="btn2">Button 2</Button>
        <Button Name="btn3">Button 3</Button>
    </StackPanel>
</Window>
```

也可以在代码中关联附加事件，但是需要使用UIElement.AddHandle方法，而不能使用+=运算符的方式。具体实现代码如下所示：

四、WPF事件生命周期

WPF事件生命周期起始和WinForm中类似。下面详细解释下WPF中事件的生命周期。

FrameworkElement类实现了ISupportInitialize.aspx)接口，该接口提供了两个用于控制初始化过程的方法。第一个是BeginInit.aspx)方法，在实例化元素后立即调用该方法。BeginInit方法被调用之后，XAML解析器设置所有元素的属性并添加内容。第二个是EndInit方法，当初始化完成后，该方法被调用。此时引发Initialized.aspx)事件。更准确地说，XAML解析器负责调用BeginInit方法和EndInit方法。

当创建窗口时，每个元素分支都以自下而上的方式被初始化。这意味着位于深层的嵌套元素在它们容器之前先被初始化。当引发初始化事件时，可以确保元素树中当前元素以下的元素已经全部完成了初始化。但是，包含当前元素的容器还没有初始化，而且也不能假设窗口的其他部分也已经完成初始化了。在每个元素都完成初始化之后，还需要在它们的容器中进行布局、应用样式，如果需要的话还会进行数据绑定。

一旦初始化过程完成后，就会引发Loaded事件。Loaded事件和Initialized事件的发生过程相反。意思就是说，包含所有元素的窗口首先引发Loaded事件，然后才是更深层次的嵌套元素。当所有元素都引发了Loaded事件之后，窗口就变得可见了，并且元素都已被呈现。下图列出了部分生命周期事件。

名　称	说　明
SourceInitialized	当取得窗口的 HwndSource 属性时(但是在窗口可见之前)发生。HwndSource 是窗口句柄，如果调用 Win32 API 中的遗留函数，就可能需要使用该句柄
ContentRendered	在窗口第一次呈现结束后立即发生。对于执行任何可能会影响窗口可视外观的操作，这不是一个好位置(改用 Loaded 事件)，否则将会强制进行第二次呈现。然而，ContentRendered 事件表明窗口已经完全可见，并且已经准备好接收输入
Activated	当用户切换到该窗口时发生(例如，从应用程序的其他窗口或从其他应用程序切换到该窗口)。当窗口第一次加载时也会引发 Activated 事件。从概念上讲，窗口的 Activated 事件相当于控件的 GotFocus 事件
Deactivated	当用户从该窗口切换到其他窗口时发生(例如，切换到应用程序的其他窗口或切换到其他应用程序)。当用户关闭窗口时该事件也会发生，该事件在 Closing 事件之后但是在 Closed 事件之前发生。从概念上讲，窗口的 Deactivated 事件相当于控件的 LostFocus 事件
Closing	当关闭窗口时发生，不管是用户关闭窗口还是通过代码调用 Window.Close( )方法或调用 Application.Shutdown( )方法关闭窗口。Closing 事件提供了取消操作并保持打开状态的机会，具体通过将 CancelEventArgs.Cancel 属性设置为 true 实现该目标。但是，如果是因为用户关闭或注销计算机而导致应用程序被关闭，那么就不能接收到 Closing 事件。为了解决这种情况，需要处理将在第 7 章中描述的 Application.SessionEnding 事件
Closed	当窗口已经关闭后发生。但是，这时仍然可以访问元素对象，当然是在 Unloaded 事件还没有发生之前。在此，可以执行一些清理工作，向永久存储位置(如配置文件或者 Windows 注册表)写入设置信息等

五、小结

到这里，WPF路由事件的内容就介绍结束了，本文首先介绍了路由事件的定义，接着介绍了三种路由事件，WPF包括直接路由事件、冒泡路由事件和隧道路由事件，最后介绍了WPF事件的生命周期。在后面一篇文章将介绍WPF中的元素绑定。

本文所有源代码下载:WPFRouteEventDemo.zip

WPF快速入门系列(4)——深入解析WPF绑定

一、引言

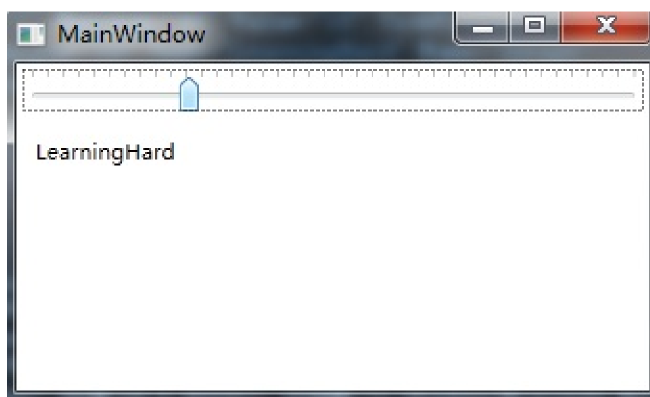
WPF绑定使得原本需要多行代码实现的功能，现在只需要简单的XAML代码就可以完成之前多行后台代码实现的功能。WPF绑定可以理解为一种关系，该关系告诉WPF从一个源对象提取一些信息，并将这些信息来设置目标对象的属性。目标属性总是依赖属性。然而，源对象可以是任何内容，可以是一个WPF元素、或ADO.NET数据对象或自定义的数据对象等。下面详细介绍了WPF绑定中的相关知识点。

二、绑定元素对象

2.1 如何实现绑定元素对象

这里首先介绍绑定最简单的情况——绑定元素对象，即数据源是一个WPF元素对象并且源属性是依赖属性。由于依赖属性具有内置的更改通知支持，因此，当在源对象中改变依赖属性的值时，会立即更新目标对象中的绑定属性。下面通过一个简单的例子来演示下如何绑定元素对象。具体的XAML代码(这里不需要后台代码)如下所示：

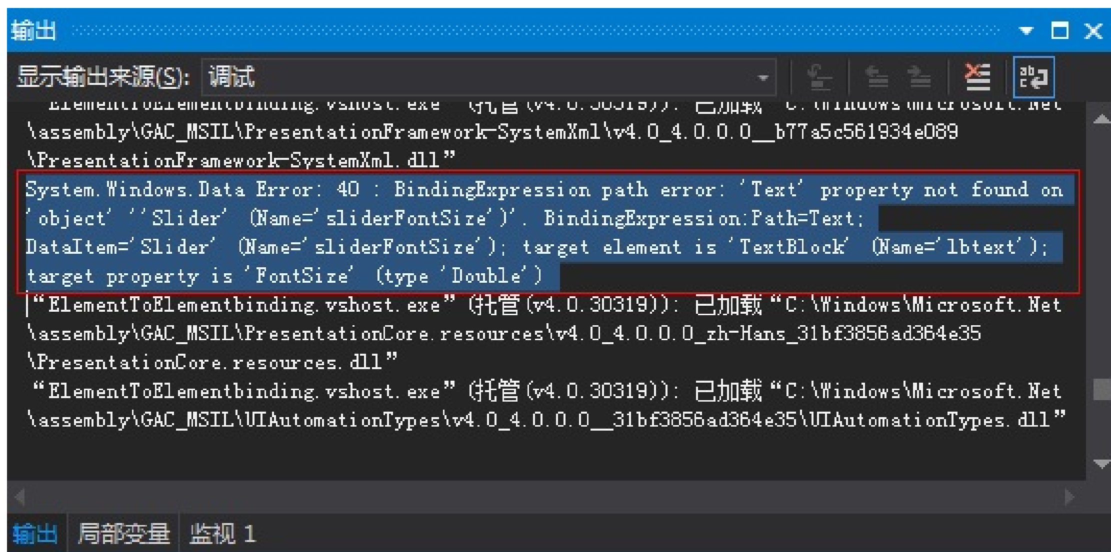
在上面XAML代码中，TextBlock控件的FontSize属性绑定了Slider控件的Value属性，感觉说绑定有点拗口，你可以直接理解为TextBlock的FontSize属性的值来自与Slider控件的Value值，由于源属性Value是依赖属性，具体内置的更改通知功能，所以Slider控件Value值的改变，直接影响TextBlock控件FontSize的值。正如我们分析的那样，实际运行结果也是如此，运行结果如下图所示：



当移动上图中Slider控件上的游标时，下面的文本字体大小也会跟着一起改变。具体效果这里就不贴图了，大家可以自行尝试。从中可以看到WPF绑定的强大了吧，如果放到以前WinForm开发中，你需要监听Slider的ValueChanged事件，然后在事件处理程序中去动态改变文本的字体大小。

这里Path除了可以直接绑定属性之外，还可以绑定属性的属性，如FontFamily.Source，也可以指向属性使用的索引器，如Content.Children[0]。当然你也可以执行多层次的路径，如指向属性的属性的属性等。

另外，如果绑定失败时，WPF不会引发异常来告知绑定失败的原因。例如，指定的元素或属性不存在，此时不会收到任何提示，只是在目标属性不能显示数据罢了。然而在调试模式下，你可以在输出窗口来查看绑定失败的信息，例如，在上面XAML代码，我们绑定Slider控件一个不存在的属性，如Text属性，此时在Output窗口中就会看到如下信息：



2.2 绑定模式

绑定的一个最大的特点就是源属性改变时，目标属性会自动地更新。然而上面的示例有一个问题，即目标对象的改变不会自动更新源对象的属性。通过下面的例子可以看出这个问题所在。此时XAML代码修改为：

```
<Window x:Class="WPFBindingDemo.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/present
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="400">
    <StackPanel>
        <Slider Name="sliderFontSize" Margin="3"
            Minimum="1" Maximum="40" Value="10" TickFrequency="1" T
        <TextBlock Margin="10" Text="LearningHard" Name="lbtext"
                   FontSize="{Binding ElementName=sliderFontSize, F

        <!--在按钮的Click事件处理程序中去改变目标对象的FontSize的值-->
        <StackPanel Orientation="Horizontal">
        <Button Margin="10" Padding="5" Click="cmd_SetSmall">Set to
        <Button Margin="10" Padding="5" Click="cmd_SetNormal">Set t
        <Button Margin="10" Padding="5" Click="cmd_SetLarge">Set to
        </StackPanel>
    </StackPanel>
</Window>
```

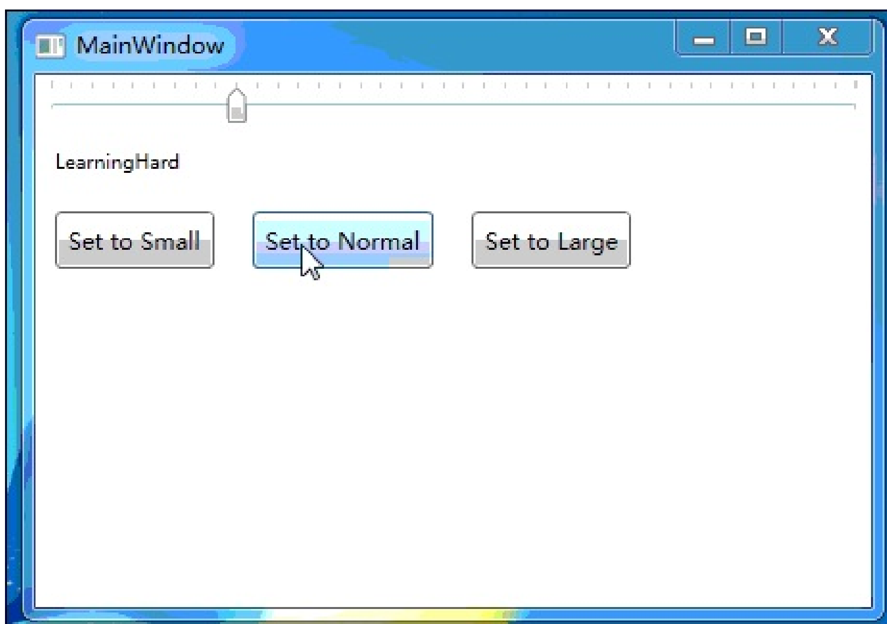
此时后台C#代码如下所示：

```
        private void cmd_SetSmall(object sender, RoutedEventArgs e)
        {
            // 仅仅在双向模式下工作
            lbtext.FontSize = 5;
        }

        private void cmd_SetNormal(object sender, RoutedEventArgs e
        {
            sliderFontSize.Value = 20;
        }

        private void cmd_SetLarge(object sender, RoutedEventArgs e)
        {
            // 仅仅在双向模式下工作
            lbtext.FontSize = 40;
        }
```

具体的运行效果如下图所示：



从上图可以看到，当在后台更改TextBlock的FontSize属性值，而Slider的Value值却没有进行更新。此时，你肯定会想问，能不能实现目标属性的更变也会自动改变绑定中源属性的机制呢？因为这样就不会显得那样呆板了，然而，你想到的了WPF团队肯定也想到了，WPF支持双向绑定，即从源到目标以及目标到源，要支持双向绑定，只需要设置Binding对象的Mode.aspx)属性为TwoWay即可，修改后的XAML代码为：

```
<StackPanel>
        <Slider Name="sliderFontSize" Margin="3"
            Minimum="1" Maximum="40" Value="10" TickFrequency="1" T
        <TextBlock Margin="10" Text="LearningHard" Name="lbtext"
                   FontSize="{Binding ElementName=sliderFontSize, P

        <!--在按钮的Click事件处理程序中去改变目标对象的FontSize的值-->
        <StackPanel Orientation="Horizontal">
        <Button Margin="10" Padding="5" Click="cmd_SetSmall">Set to
        <Button Margin="10" Padding="5" Click="cmd_SetNormal">Set t
        <Button Margin="10" Padding="5" Click="cmd_SetLarge">Set to
        </StackPanel>
    </StackPanel>
```

Mode属性除了可以设置OneWay，TwoWay值外，还可以设置Default、OneTime和OneWayToSource，关于这些值更详细的介绍请自行参考MSDN：http://msdn.microsoft.com/zh-cn/library/system.windows.data.bindingmode(v=vs.110).aspx.aspx)。

另外，除了可以在XAML中通过Binding标记地方式声明绑定外，还可以使用代码方式动态创建绑定。如上面的例子中代码创建绑定的实现代码如下所示：

还可以通过使用BindingOperations.aspx)类的ClearBinding.aspx)方法来移除数据绑定。还可以使用ClearAllBindings.aspx)移除一个元素的所有数据绑定。

2.3 绑定更新

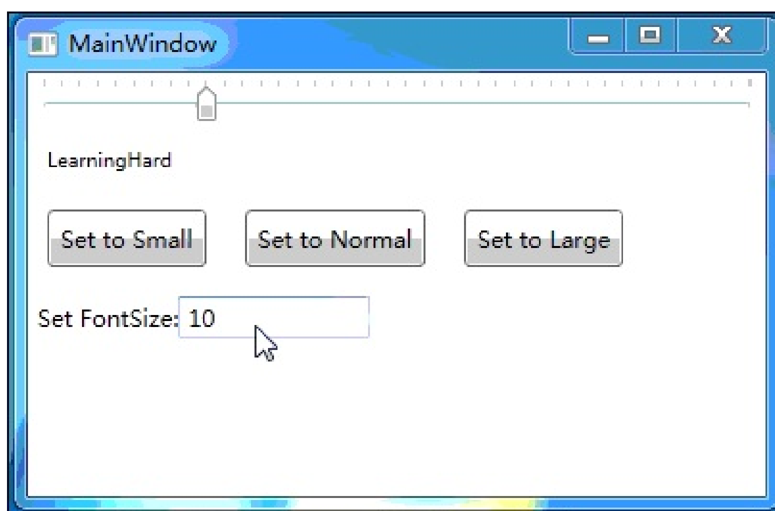
在上面的例子中，还存在另一个问题，当通过在文本框中输入内容来改变显示的字体尺寸时，此时什么事情都不会发生，知道使用tab键将焦点转移到另外一个控件时，才会应用对应的改变。此时XAML代码如下所示：

```
<Window x:Class="WPFBindingDemo.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/present
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="400">
    <StackPanel>
        <Slider Name="sliderFontSize" Margin="3"
            Minimum="1" Maximum="40" Value="10" TickFrequency="1" T
        <TextBlock Margin="10" Text="LearningHard" Name="lbtext"
                   FontSize="{Binding ElementName=sliderFontSize, P

        <!--在按钮的Click事件处理程序中去改变目标对象的FontSize的值-->
        <StackPanel Orientation="Horizontal">
        <Button Margin="10" Padding="5" Click="cmd_SetSmall">Set to
        <Button Margin="10" Padding="5" Click="cmd_SetNormal">Set t
        <Button Margin="10" Padding="5" Click="cmd_SetLarge">Set to
        </StackPanel>

        <!--添加一个输入文本框来设置文本字体大小进行测试问题-->
        <StackPanel Orientation="Horizontal" Margin="5">
            <TextBlock VerticalAlignment="Center">Set FontSize:</Te
            <TextBox Text="{Binding ElementName=lbtext, Path=FontSi
        </StackPanel>
    </StackPanel>
</Window>
```

后台代码实现与前面的一样，此时运行的效果如下图所示：



为了明白导致这个问题的原因，这里需要深入分析下绑定表达式。当使用OneWay或TwoWay绑定时，改变后的值会立即从源传播到目标。对于滑动条，然而，从目标到源传播未必会立即发生。因为，它们的行为是由Binding.UpdateSourceTrigger.aspx)属性控制，该属性可以使用下图列出的某个值。注意，**UpdateSourceTrigger**属性值并不影响目标的更新方式，它仅仅控制**TwoWay**模式或**OneWayToSource**模式的绑定更新源的方式。而文本框正是使用LostFocus方式从目标向源进行更新的。

	成员名称	说明
	Default	绑定目标属性的默认 UpdateSourceTrigger 值。 大多数依赖项属性的默认值都为 PropertyChanged，而 Text 属性的默认值为 LostFocus。 确定依赖项属性的默认 UpdateSourceTrigger 值的编程方法是使用 GetMetadata 来获取属性的属性元数据，然后检查 DefaultUpdateSourceTrigger 属性的值。
	Explicit	仅在调用 UpdateSource 方法时更新绑定源。
	LostFocus	当绑定目标元素失去焦点时，更新绑定源。
	PropertyChanged	当绑定目标属性更改时，立即更新绑定源。

既然，找出了导致原因，此时可以对XAML代码进行修改，使得当用于在文本框中输入内容时将变化应用于字体尺寸，具体改变部分的XAML代码为：

另外，需要注意的是，TextBox的Text属性的默认行为是LostFocus，这是因为当用于输入内容时，文本框中文本会不断变化，从而引起多次更新。所以PropertyChanged模式可能会使应用程序运行更缓慢，所以LostFocus默认行为可以说是合理的。

要完全控制源对象的更新时机，也可以选择UpdateSourceTrigger.Explicit模式。此时就需要额外编写代码手动触发更新，此时可以添加一个Apply按钮，并在按钮的Click事件处理程序中调用BindingExpression.UpdateSource.aspx?query=UpdateSource)方法触发立即刷新并更新字体大小的操作。具体的实现代码如下所示：

三、绑定非元素对象

上面都是介绍如何链接两个元素的绑定，但是在数据驱动的应用程序中，更常见的情况是创建从一个对象中提起数据的绑定表达式。不过希望绑定的信息必须存储在一个公有属性中。因为WPF绑定不能获取私有信息或公有字段。

当绑定一个非元素对象时，不能使用Binding.ElementName属性，但可以使用以下属性中的一个：

- Source.aspx)——该属性是指向源对象的引用，即提供数据的对象。
- RelativeSource.aspx)——该属性使用RelativeSource对象指定绑定源的相对位置，默认值为null。
- DataContext属性——如果没有使用Source或RelativeSource属性指定一个数据源，WPF会从当前元素开始在元素树中向上查找。检查每个元素的DataContext属性，并使用第一个非空的DataContext属性。当然你也可以自己设置DataContext属性。

下面通过一个例子来演示下如何绑定到非元素对象。下面的演示如何使用DataContext属性来绑定一个自定义对象的属性。首先自定义一个实现了INotifyPropertyChanged.aspx)接口的类。这个接口是为了发出属性更改的通知，即实现了这个接口将会实现当源对象的公共属性发生改变时，该属性的值会立即响应到界面上显式。当然不实现这个接口的对象也可以绑定控件中，只要被绑定是公有属性就可以。具体的实现代码如下所示：

```
 1 using System.ComponentModel;
 2 
 3 namespace WPFBindingDemo
 4 {
 5     public class Student:INotifyPropertyChanged
 6     {
 7         private int m_ID;
 8         private string m_StudentName;
 9         private double m_Score;
10 
11         public int ID
12         {
13             get { return m_ID; }
14             set
15             {
16                 if (value != m_ID)
17                 {
18                     m_ID = value;
19                     Notify("ID");
20                 }
21             }
22         }
23 
24         public string StudentName
25         {
26             get { return m_StudentName; }
27             set
28             {
29                 if (value != m_StudentName)
30                 {
31                     m_StudentName = value;
32                     Notify("StudentName");
33                 }
34             }
35         }
36 
37         public double Score
38         {
39             get { return m_Score; }
40             set
41             {
42                 if (value != m_Score)
43                 {
44                     m_Score = value;
45                     Notify("Score");
```

```
46                 }
47             }
48         }
49
50         public event PropertyChangedEventHandler PropertyChangec
51         private void Notify(string propertyName)
52         {
53             if (PropertyChanged != null)
54             {
55                 this.PropertyChanged(this, new PropertyChangedEv
56             }
57         }
58     }
59 }
```

既然源数据对象以准备好了，自然接下来就是去设计WPF界面来让控件来绑定这个源对象了，具体的XAML代码如下所示：

```
<Window x:Class="WPFBindingDemo.BindingToCollection"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/present
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="BindingToCollection" Height="300" Width="300">
    <StackPanel Margin="50">
        <StackPanel Orientation="Horizontal" Margin="10">
            <TextBlock Text="学号:"  />
            <TextBlock Text="{Binding Path=ID}" Width="100"/>
        </StackPanel>
        <StackPanel Orientation="Horizontal" Margin="10">
            <TextBlock Text="姓名:"  />
            <TextBlock Text="{Binding Path=StudentName}" Width="10(
        </StackPanel>
        <StackPanel Orientation="Horizontal" Margin="10">
            <TextBlock Text="分数:"  />
            <TextBlock Text="{Binding Path=Score}" Width="100"/>
        </StackPanel>

        <StackPanel Orientation="Horizontal" Margin="10">
            <Button Content="改变姓名" Name="changeName" Click="chan
            <Button Content="改变分数" Name="changeScore" Margin="2(
        </StackPanel>
    </StackPanel>
</Window>
```

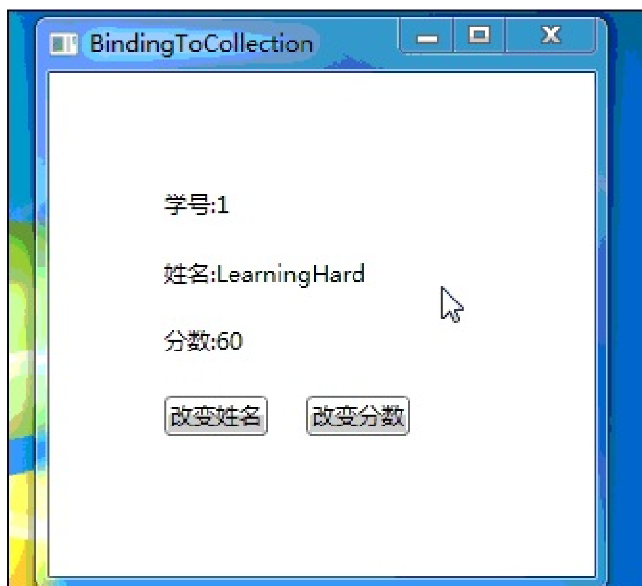
对应的后台代码逻辑如下所示：

```
public partial class BindingToCollection : Window
    {
        private Student m_student;
        public BindingToCollection()
        {
            InitializeComponent();
            m_student = new Student() { ID = 1, StudentName = "Lear
            // 设置Window对象的DataContext属性
            this.DataContext = m_student ;
        }

        private void ChangeScore_Click(object sender, RoutedEventAr
        {
            m_student.Score = 90;
        }

        private void changeName_Click_1(object sender, RoutedEventA
        {
            m_student.StudentName = "Learning";
        }
    }
```

完成了示例所有代码的编写之后，下面具体看看示例的运行效果，看看是否可以成功完成绑定并源对象的属性的更改会立即反应到界面中，具体的效果图如下所示：



从上图示例的演示动画效果可以看出，上面的代码确实实现我们预期的功能。从上面代码可以看出，我们并没有对每个控件单独设置它的Source属性，而是直接设置了Window对象的DataContext属性。这样绑定的控件发现没有设置source属性或RelativeSource属性，就会从元素树中查找DataContex属性不为null的值来作为自己的DataContext。通过这样的方式可以省去重复在多个控件中设置相同的DataContext属性。

这里只是演示了绑定单个数据对象的情况，就如之前所说的，数据源还可以是XAML文件，ADO.NET数据对象、集合等，这里就不一一实现了，只要了解具体思路，具体问题具体搜索解决就好了。这里给出两个非常的好例子。

Simple Demo of Binding to a Database in WPF using LINQ-SQL

How to Perform WPF Data Binding Using LINQ to XML

四、提高大列表的性能

如果绑定的数据源具有大量记录时，此时就需要考虑性能的问题了。然而，幸运的是，WPF很多列表控件都已经帮我们做好了相应的支持，这里只是提出来让大家知道这点。

对于大列表显示性能问题，WPF做了以下几种支持：

- UI虚拟化——UI虚拟化是列表仅为当前显示项创建容器对象的一种技术，例如，如果有一个具有5万条记录的列表，但是可见区域只能包含30条记录，ListBox控件只创建30个ListBoxItem对象。如果ListBox控件不支持UI虚拟化的话，它将需要生成全部5万个ListBoxItem对象，这显然需要占用更多的内存。并且分配这些对象的时间用户明显可以感觉到，这就带来了非常不好的用户体验。WPF中UI虚拟化是通过VirtualizingStackPanel容器实现的。像ListBox、ListView和DataGrid都自动使用VirtualizingStackPanel面板布局它们的子元素，所以，这些控件都默认支持虚拟化功能。然而，ComboBox需要支持虚拟化支持，必须明确提供新的ItemPanelTemplate添加虚拟化支持，具体实现如下所示：

```
<ComboBox>
   <ComboBox.ItemsPanel>
       <ItemsPanelTemplate>
          ** <VirtualizingStackPanel></VirtualizingStackPanel>**
       </ItemsPanelTemplate>
   </ComboBox.ItemsPanel>
</ComboBox>
```

TreeView控件也支持虚拟化，但它在默认情况下，关闭了该支持，你需要显式启用该特性，具体使用的启用代码如下所示：

- 项目容器再循环——WPF 3.5 SP1使用项目容器再循环改进了虚拟化。通常支持虚拟化的列表在滚动时，控件不断地创建新的项目容器对象来保存新的可见项。例如，当具有5万个项的ListBox控件，在滚动时，ListBox需要重新生成新的ListBoxItem对象。但是如果启用了项目容器再循环，ListBox控件会保存少量ListBoxItem对象存活，当滚动时，将新数据加到这些之前的ListBoxItem对象，从而重复使用它们。具体支持代码如下所示

项目容器再循环提供了滚动性能，并降低了内存消耗量，因为垃圾回收器不需要查找旧对象进行回收。为了确保向后兼容，除了DataGrid之后的所有列表控件默认都禁用该特性，如需支持，需要显式启用。

- 延迟滚动——为了进一步提供滚动性能，可以开启延迟滚动功能。使用延迟滚动，当用户在滚动条上拖动滚动滑块时不更新列表显示，只有用户释放了滚动滑块时才刷新。当使用常规滚动时，在拖动的同时会刷新列表，使列表显示正在改变的位置。这个特性也需要显式启用，启用代码如下：

显然，需要在响应性和易用性之间平衡。如果有一个复杂的模板和大量数据，对于提高速度可能会选择使用延迟滚动特性，但当用户需要在滚动时查看目前滚动位置，则就可以不启用该特性。

上面介绍了这么多，其实提供列表控件的性能主要在两方面：UI虚拟化提高了列表项初始化的时间，因为UI虚拟化支持一次性不初始化所有项，而在滚动是自动创建新的项。项目容器再循环和延迟滚动提高了滚动性能。

另外WPF绑定还有两个知识点：数据验证和数据转换，对于数据验证与Asp.net中验证类似，都是为了保证输入数据的合法性，而数据转换指的是在源数据绑定到目标依赖属性之前要做对应的转换，例如WPF显示人民币都需要显示一个￥符号，但是如果数据源的内容只是“120”这样的字符串怎么办呢？这时候就可以通过数据转换在绑定之前，把数据源的值转换成显示所需要的格式。对于这两个知识点，我觉得在遇到问题时再去学就好了，因为我们已经明白了解决问题的思路了。所以，在快速入门系列中不想太深入的介绍这两个知识点，以使大家可以快速掌握WPF要领。这里给出几个学习链接：

数据绑定概述.aspx) WPF Data Binding - Part 1

WPF Simple Data Converter Example

如何：实现绑定验证.aspx)

五、小结

到这里，这篇博文的内容就介绍结束了，时间不知不觉的已经2点多了。下面一篇博文将分享WPF命令的内容。

本文所有源码下载：WPFBindingDemo.zip

WPF快速入门系列(5)——深入解析WPF命令

一、引言

WPF命令相对来说是一个崭新的概念，因为命令对于之前的WinForm根本没有实现这个概念，但是这并不影响我们学习WPF命令，因为设计模式中有命令模式，关于命令模式可以参考我设计模式的博文：http://www.cnblogs.com/zhili/p/CommandPattern.html。命令模式的要旨在于把命令的发送者与命令的执行者之间的依赖关系分割开了。对此，WPF中的命令也是一样的，**WPF**命令使得命令源(即命令发送者，也称调用程序)和命令目标（即命令执行者，也称处理程序）分离。现在是不是感觉命令是不是亲切了点了呢？下面就详细分享下我对WPF命令的理解。

二、命令是什么呢？

上面通过命令模式引出了WPF命令的要旨，那在WPF中，命令是什么呢？对于程序来说，命令就是一个个任务，例如保存，复制，剪切这些操作都可以理解为一个个命令。即当我们点击一个复杂按钮时，此时就相当于发出了一个复制的命令，即告诉文本框执行一个复杂选中内容的操作，然后由文本框控件去完成复制的操作。在这里，复杂按钮就相当于一个命令发送者，而文本框就是命令的执行者。它们之间通过命令对象分割开了。如果采用事件处理机制的话，此时调用程序与处理程序就相互引用了。

所以对于命令只是从不同角度理解问题的一个词汇，之前理解点击一个按钮，触发了一个点击事件，在WPF编程中也可以理解为触发了一个命令。说到这里，问题又来了，WPF中既然有了命令了？那为什么还需要路由事件呢？对于这个问题，我的理解是，事件和命令是处理问题的两种方式，它们之间根本不存在冲突的，并且WPF命令中使用了路由事件。所以准确地说WPF命令应该是路由命令。那为什么说WPF命令是路由的呢？这个疑惑将会在WPF命令模型介绍中为大家解答。

另外，WPF命令除了使命令源和命令目标分割的优点外，它还具有另一个优点：

- 使得控件的启用状态和相应的命令状态保持同步，即命令被禁用时，此时绑定命令的控件也会被禁用。

三、WPF命令模型

经过前面的介绍，大家应该已经命令了WPF命令吧。即命令就是一个操作，任务。接下来就要详细介绍了WPF命令模型了。 WPF命令模型具有4个重要元素：

- 命令——命令表示一个程序任务，并且可跟踪该任务是否能被执行。然而，命令实际上不包含执行应用程序的代码，真正处理程序在命令目标中。
- 命令源——命令源触发命令，即命令的发送者。例如Button、MenuItem等控件都是命令源，单击它们都会执行绑定的命令。

- 命令目标——命令目标是在其中执行命令的元素。如Copy命令可以在TextBox控件中复制文本。
- 命令绑定——前面说过，命令是不包含执行程序的代码的，真正处理程序存在于命令目标中。那命令是怎样映射到处理程序中的呢？这个过程就是通过命令绑定来完成的，命令绑定完成的就是红娘牵线的作用。

WPF命令模型的核心就在于ICommand.aspx)接口了，该接口定义命令的工作原理。该接口的定义如下所示：

```
public interface ICommand
{
    // Events
    event EventHandler CanExecuteChanged;

    // Methods
    bool CanExecute(object parameter);

    void Execute(object parameter);
}
```

该接口包括2个方法和一个事件。CanExecute方法返回命令的状态——指示命令是否可执行，例如，文本框中没有选择任何文本，此时Copy命令是不用的，CanExecute则返回为false。

Execute方法就是命令执行的方法，即处理程序。当命令状态改变时，会触发CanExecuteChanged事件。

当自定义命令时，不会直接去实现ICommand接口。而是使用RoutedCommand.aspx)类，该类实是WPF中唯一现了ICommand接口的类。所有WPF命令都是RoutedCommand类或其派生类的实例。然而程序中处理的大部分命令不是RoutedCommand对象，而是RoutedUICommand对象。RoutedUICommand.aspx)类派生与RoutedCommand类。

接下来介绍下为什么说WPF命令是路由的呢？实际上，RoutedCommand上Execute和CanExecute方法并没有包含命令的处理逻辑，而是将触发遍历元素树的事件来查找具有CommandBinding的对象。而真正命令的处理程序包含在CommandBinding的事件处理程序中。所以说WPF命令是路由命令。该事件会在元素树上查找CommandBinding对象，然后去调用CommandBinding的CanExecute和Execute来判断是否可执行命令和如何执行命令。那这个查找方向是怎样的呢？对于位于工具栏、菜单栏或元素的FocusManager.IsFocusScope.aspx)设置为"true"是从元素树上根元素(一般指窗口元素)向元素方向向下查找，对于其他元素是验证元素树根方向向上查找。

WPF中提供了一组已定义命令，命令包括以下类：ApplicationCommands, NavigationCommands, MediaCommands, EditingCommands, and the ComponentCommands.">ApplicationCommands.aspx)、NavigationCommands.aspx)、MediaCommands.aspx)、EditingCommands.aspx) 以及ComponentCommands.aspx)。 Cut, BrowseBack and BrowseForward, Play,

Stop, and Pause.">这些类提供诸如 Cut.aspx)、BrowseBack.aspx)、BrowseForward.aspx)、Play.aspx)、Stop.aspx) 和 Pause.aspx) 等命令。

四、使用命令

前面都是介绍了一些命令的理论知识，下面介绍了如何使用WPF命令来完成任务。XAML具体实现代码如下所示：

```
1 <Window x:Class="WPFCommand.MainWindow"
2         xmlns="http://schemas.microsoft.com/winfx/2006/xaml/pres
3         xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4         Title="MainWindow" Height="200" Width="300">
5     <!--定义窗口命令绑定，绑定的命令是New命令，处理程序是NewCommand-->
6     <Window.CommandBindings>
7         <CommandBinding Command="ApplicationCommands.New" Execut
8     </Window.CommandBindings>
9 
10     <StackPanel>
11         <Menu>
12             <MenuItem Header="File">
13                 <!--WPF内置命令都可以采用其缩写形式-->
14                 <MenuItem Command="New"></MenuItem>
15             </MenuItem>
16         </Menu>
17 
18         <!--获得命令文本的两种方式-->
19         <!--直接从静态的命令对象中提取文本-->
20         <Button Margin="5" Padding="5" Command="ApplicationComma
21 
22         <!--使用数据绑定，获得正在使用的Command对象，并提取其Text属性--
23         <Button Margin="5" Padding="5" Command="ApplicationComma
24         <Button Margin="5" Padding="5" Visibility="Visible" Clic
25     </StackPanel>
26 </Window>
```

其对应的后台代码实现如下所示：

```
public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();

            //// 后台代码创建命令绑定
            //CommandBinding bindingNew = new CommandBinding(Applic
            //bindingNew.Executed += NewCommand;
            //// 将创建的命令绑定添加到窗口的CommandBindings集合中
            //this.CommandBindings.Add(bindingNew);
        }

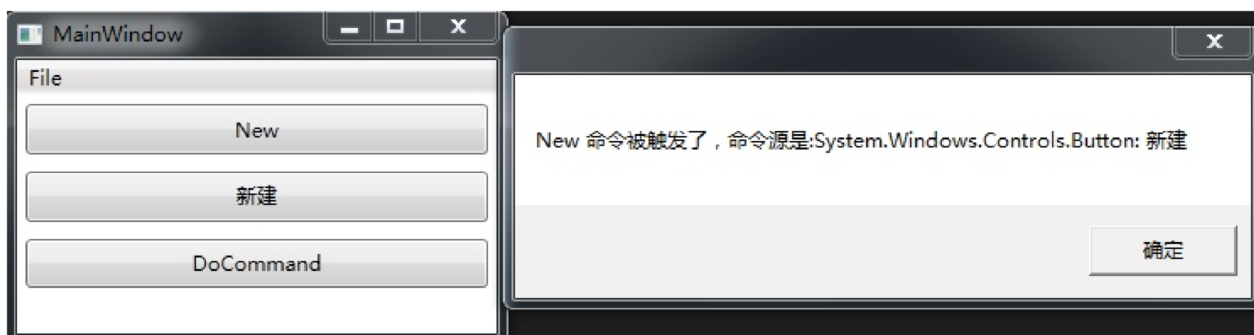
        private void NewCommand(object sender, ExecutedRoutedEventA
        {
            MessageBox.Show("New 命令被触发了，命令源是:" + e.Source.T
        }

        private void cmdDoCommand_Click(object sender, RoutedEventA
        {
            // 直接调用命令的两种方式
            ApplicationCommands.New.Execute(null, (Button)sender);

            //this.CommandBindings[0].Command.Execute(null);
        }

    }
```

上面程序的运行结果如下图所示：



五、自定义命令

在开发过程中，自然少不了自定义命令来完成内置命令所没有提供的任务。下面通过一个例子来演示如何创建一个自定义命令。

首先，定义一个Requery命令，具体的实现如下所示：

```
public class DataCommands
    {
        private static RoutedUICommand requery;
        static DataCommands()
        {
            InputGestureCollection inputs = new InputGestureCollect
            inputs.Add(new KeyGesture(Key.R, ModifierKeys.Control,
            requery = new RoutedUICommand(
              "Requery", "Requery", typeof(DataCommands), inputs);
        }

        public static RoutedUICommand Requery
        {
            get { return requery; }
        }
    }
```

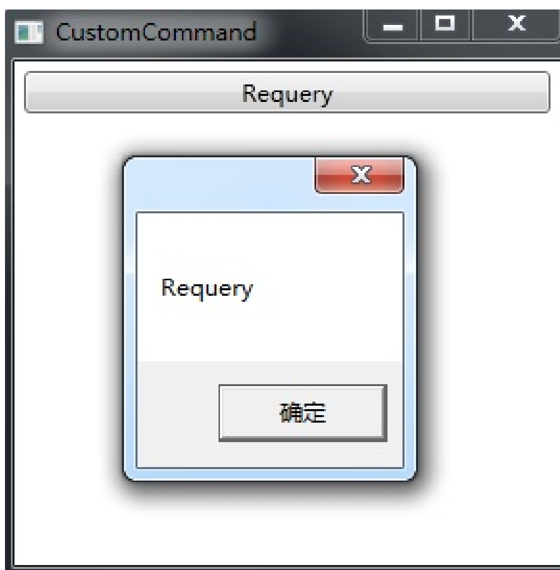
上面代码实现了一个Requery命令，为了演示效果，我们需要把该命令应用到XAML标签上，具体的XAML代码如下所示：

```
<!--要使用自定义命令，首先需要将.NET命名空间映射为XAML名称空间，这里映射的命名
    <Window x:Class="WPFCommand.CustomCommand"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/present
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

        xmlns:local="clr-namespace:WPFCommand"
        Title="CustomCommand" Height="300" Width="300" >

    <Window.CommandBindings>
        <!--定义命令绑定-->
        <CommandBinding Command="local:CustomCommands.Requery" Exec
    </Window.CommandBindings>
    <StackPanel>
        <!--应用命令-->
        <Button Margin="5" Command="local:CustomCommands.Requery" C
    </StackPanel>
</Window>
```

接下来，看看程序的运行效果，具体的运行结果如下图所示：



六、实现可撤销的命令程序

WPF命令模型缺少的一个特征就是Undo命令，尽管提供了一个ApplicationCommands.Undo命令，但是该命令通常被用于编辑控件，如TextBox控件。如果希望支持应用程序范围内的Undo操作，就需要在内部跟踪以前的命令，并且触发Undo操作时还原该命令。这个实现原理就是保持用一个集合对象保存之前所有执行过的命令，当触发Undo操作时，还要上一个命令的状态。这里除了需要保存执行过的命令外，还需要保存触发命令的控件以及状态，所以我们需要抽象出一个类来保存这些属性，我们取名这个类为CommandHistoryItem。为了保存命令和命令的状态，自然就需要在完成命令之前进行保存，所以自然联想到是否有Preview之类的事件呢？实际上确实有，这个事件就是PreviewExecutedEvent，所以我们需要在窗口加载完成后把这个事件注册到窗口上，这里在触发这个事件的时候就可以保存即将要执行的命令、命令源和命令源的内容。另外，之前的命令自然需要保存到一个列表中，这里使用ListBox控件作为这个列表，如果不希望用户在界面上看到之前的命令列表的话，也可以使用List等集合容器。

上面讲解完了主要实现思路之后，下面我们梳理下实现思路：

1. 抽象一个**CommandHistoryItem**来保存命令相关的属性。
2. 注册**PreviewExecutedEvent**事件，为了在命令执行完之前保存命令、命令源以及命令源当前的状态。
3. 在**PreviewExecutedEvent**事件处理程序中，把命令相关属性添加到**ListBox**列表中。
4. 当执行撤销操作时，可以从**ListBox.Items**列表中取出上一个执行的命令进行恢复之前命令的状态。

有了上面的实现思路之后，实现这个可撤销的命令程序也就是码代码的过程了。具体的后台代码实现如下所示：

```
1  public partial class CommandsMonitor : Window
2      {
3          private static RoutedUICommand undo;
4          public static RoutedUICommand Undo
```

```
 5          {
 6              get { return CommandsMonitor.undo; }
 7          }
 8
 9          static CommandsMonitor()
10          {
11              undo = new RoutedUICommand("Undo", "Undo", typeof(C
12          }
13
14          public CommandsMonitor()
15          {
16              InitializeComponent();
17              // 按下菜单栏按钮时，PreviewExecutedEvent事件会被触发2次
18              // 一次是菜单栏按钮本身，一次是目标源触发命令的执行，所以在C
19              this.AddHandler(CommandManager.PreviewExecutedEvent
20          }
21
22          public void CommandExecuted(object sender, ExecutedRout
23          {
24              // 过滤掉命令源是菜单按钮的，因为我们只关心Textbox触发的命令
25              if (e.Source is ICommandSource)
26                  return;
27              // 过滤掉Undo命令
28              if (e.Command == CommandsMonitor.Undo)
29                  return;
30
31              TextBox txt = e.Source as TextBox;
32              if (txt != null)
33              {
34                  RoutedCommand cmd = e.Command as RoutedCommand;
35                  if (cmd != null)
36                  {
37                      CommandHistoryItem historyItem = new Comman
38                      {
39                          CommandName = cmd.Name,
40                          ElementActedOn = txt,
41                          PropertyActedOn = "Text",
42                          PreviousState = txt.Text
43                      };
44
45                      ListBoxItem item = new ListBoxItem();
46                      item.Content = historyItem;
47                      lstHistory.Items.Add(item);
48                  }
49
50              }
51          }
52
53          private void window_Unloaded(object sender, RoutedEvent
54          {
55              this.RemoveHandler(CommandManager.PreviewExecutedEv
56          }
57
```

```
 58         private void UndoCommand_Executed(object sender, Routed
 59         {
 60             ListBoxItem item = lstHistory.Items[lstHistory.Iten
 61 
 62             CommandHistoryItem historyItem = item.Content as Co
 63             if (historyItem == null)
 64             {
 65                 return;
 66             }
 67 
 68             if (historyItem.CanUndo)
 69             {
 70                 historyItem.Undo();
 71             }
 72             lstHistory.Items.Remove(item);
 73         }
 74 
 75         private void UndoCommand_CanExecuted(object sender, Car
 76         {
 77             if (lstHistory == null || lstHistory.Items.Count ==
 78             {
 79                 e.CanExecute = false;
 80             }
 81             else
 82             {
 83                 e.CanExecute = true;
 84             }
 85         }
 86     }
 87 
 88     public class CommandHistoryItem
 89     {
 90         public String CommandName { get; set; }
 91         public UIElement ElementActedOn { get; set; }
 92 
 93         public string PropertyActedOn { get; set; }
 94 
 95         public object PreviousState { get; set; }
 96 
 97         public bool CanUndo
 98         {
 99             get { return (ElementActedOn != null && PropertyAct
100         }
101 
102         public void Undo()
103         {
104             Type elementType = ElementActedOn.GetType();
105             PropertyInfo property = elementType.GetProperty(Pro
106             property.SetValue(ElementActedOn, PreviousState, nu
107         }
108     }
109 }
```

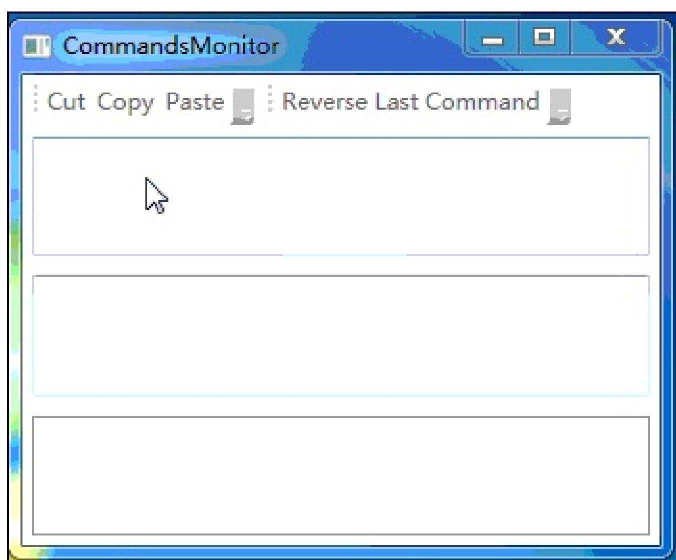
其对应的XAML界面设计代码如下所示：

```
<Window x:Class="WPFCommand.CommandsMonitor"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/present
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="CommandsMonitor" Height="300" Width="350"
        xmlns:local="clr-namespace:WPFCommand"
        Unloaded="window_Unloaded">
    <Window.CommandBindings>
        <CommandBinding Command="local:CommandsMonitor.Undo"
                        Executed="UndoCommand_Executed"
                        CanExecute="UndoCommand_CanExecuted"/>
    </Window.CommandBindings>
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"></RowDefinition>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
        </Grid.RowDefinitions>
        <ToolBarTray Grid.Row="0">
            <ToolBar>
                <Button Command="ApplicationCommands.Cut">Cut</Butt
                <Button Command="ApplicationCommands.Copy">Copy</Bu
                <Button Command="ApplicationCommands.Paste">Paste</
            </ToolBar>
            <ToolBar>
                <Button Command="local:CommandsMonitor.Undo">Revers
            </ToolBar>
        </ToolBarTray>

        <TextBox Margin="5" Grid.Row="1"
             TextWrapping="Wrap" AcceptsReturn="True">
        </TextBox>
        <TextBox Margin="5" Grid.Row="2"
             TextWrapping="Wrap" AcceptsReturn="True">
        </TextBox>

        <ListBox Grid.Row="3" Name="lstHistory" Margin="5" DisplayM
    </Grid>
</Window>
```

上面程序的运行效果如下图所示：



七、小结

到这里，WPF命令的内容就介绍结束了，关于命令主要记住命令模型四要素——命令、命令绑定、命令源和命令目标。后面继续为大家分享WPF的资源和样式的内容。

本文所有源码：WPFCommandDemo.zip

WPF快速入门系列(6)——WPF资源和样式

一、引言

WPF资源系统可以用来保存一些公有对象和样式，从而实现重用这些对象和样式的作用。而WPF样式是重用元素的格式的重要手段，可以理解样式就如CSS一样，尽管我们可以在每个控件中定义格式，但是如果多个控件都应用了多个格式的时候，我们就可以把这些格式封装成格式，然后在资源中定义这个格式，之前如果用到这个格式就可以直接使用这个样式，从而达到重用格式的手段。从中可以发现，WPF资源和WPF样式是相关的，我们经常把样式定义在资源中。

二、WPF资源详解

2.1 资源基础介绍

尽管可以在代码中创建和操作资源，但是通常都是以XAML标签的形式定义资源的。下面具体看看如何去定义一个资源，具体的XAML代码如下所示：

```
<Window x:Class="ResourceDemo.ResourceUse"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/present
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="REsource" Height="100" Width="350"
        xmlns:sys="clr-namespace:System;assembly=mscorlib">
  **  <Window.Resources>
        <!--定义一个字符串资源-->
        <sys:String x:Key="nameStr">
            LearningHard博客：http://www.cnblogs.com/zhili/
        </sys:String>
    </Window.Resources>**
    <StackPanel>
        <!--通过资源key来对资源进行使用-->
        <TextBlock Text="{StaticResource nameStr}" Margin="10"/>
    </StackPanel>
</Window>
```

每一个元素都有一个Resources属性，该属性存储了一个资源字典集合。关于资源字典将会在下面部分介绍。尽管每个元素都提供了Resources属性，但通常在窗口级别上定义资源，就如上面XAML代码所示的那样。因为每个元素都可以访问它自己的资源集合中的资源，也可以访问所有父元素的资源集合中的资源。

2.2 静态资源和动态资源区别

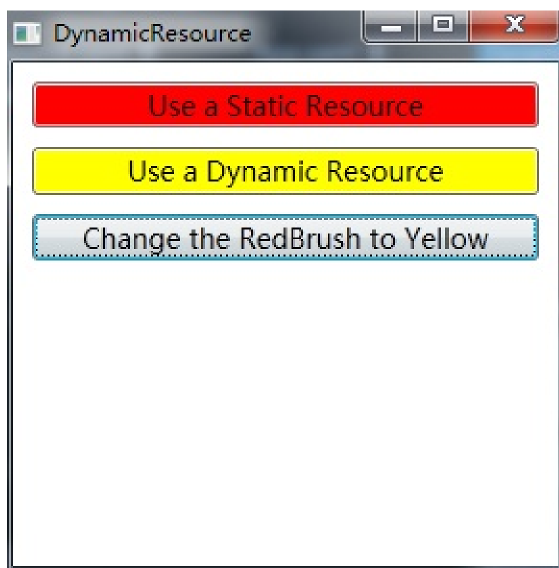
为了使用XAML标记中的资源，需要一种引用资源的方法，可以通过两个标记来进行引用资源：一个用于静态资源，另一个用于动态资源。在上面的XAML中，我们引用的方式就是静态资源的引用方式，因为我们指定了**StaticResource**。那静态资源和动态资源有什么区别呢？

对于静态资源在第一次创建窗口时，一次性地设置完毕；而对于动态资源，如果发生了改变，则会重新应用资源。下面通过一个示例来演示下他们之间的区别。具体的XAML代码如下所示：

```
<Window x:Class="ResourceDemo.DynamicResource"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/present
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="DynamicResource" Height="300" Width="300">
    <Window.Resources>
        <SolidColorBrush x:Key="RedBrush" Color="Red"></SolidColorE
    </Window.Resources>
    <StackPanel Margin="5">
    <Button Background="{StaticResource RedBrush}" Margin="5" FontS
        <Button Background="{DynamicResource RedBrush}" Margin="5"
        <Button Margin="5" FontSize="14" Content="Change the RedBru
    </StackPanel>
</Window>
```

对应改变资源按钮的后台代码如下所示：

运行上面程序，你将发现，当点击Change按钮之后，只改变了动态引用资源按钮的背景色，而静态引用按钮的背景却没有发生改变，具体效果图如下所示：



2.3 资源字典

在前面中讲到，每个Resources.aspx)属性存储着一个资源字典集合。如果希望在多个项目之间共享资源的话，就可以创建一个资源字典。资源字段是一个简单的XAML文档，该文档就是用于存储资源的，可以通过右键项目->添加资源字典的方式来添加一个资源字典文件。下面具体看下如何去创建一个资源字典。具体的XAML代码如下：

为了使用资源字典，需要将其合并到应用程序中资源集合位置，当然你也可以合并到窗口资源集合中，但是通常是合并到应用程序资源集合中，因为资源字典的目的就是在于多个窗体中共享，具体的XAML代码如下所示：

```
<Application x:Class="ResourceDemo.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/pr
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="DynamicResource.xaml">
    <Application.Resources>
        <!--合并资源字典到Application.Resources中-->
        <ResourceDictionary>
            <ResourceDictionary.MergedDictionaries>
                <ResourceDictionary Source="Generic.xaml"/>
            </ResourceDictionary.MergedDictionaries>
        </ResourceDictionary>
    </Application.Resources>
</Application>
```

那怎样使用资源字典中定义的资源呢？其使用方式和引用资源的方式是一样的，一样是通过资源的Key属性来进行引用的，具体使用代码如下所示：

```
<Window x:Class="ResourceDemo.ResourceUse"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/present
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="REsource" Height="100" Width="350"
        xmlns:sys="clr-namespace:System;assembly=mscorlib">
    <Window.Resources>
        <!--定义一个字符串资源-->
        <sys:String x:Key="nameStr">
            LearningHard博客：http://www.cnblogs.com/zhili/
        </sys:String>
    </Window.Resources>
    <StackPanel>
      **  <!--使用资源字典中定义的资源-->
        <Button  Margin="10" Background="{StaticResource blueBrush]
        <!--通过资源key来对资源进行使用-->
        <TextBlock Text="{StaticResource nameStr}" Margin="10"/>
    </StackPanel>
</Window>
```

此时的运行效果如下图所示：



前面只是介绍在当前应用程序下共享资源可以把资源字典合并到应用程序资源集合中，如果想在多个应用程序共享资源怎么办呢？最简单的方法就是在每个应用程序中拷贝一份资源字典的XAML文件，但是这样不能对版本进行控制，显然这不是一个好的办法。更好的办法是将资源字典编译到一个单独的类库程序集中，应用程序可以通过引用程序集的方式来共享资源。这样就达到了在多个应用程序中共享资源的目的。

使用这种方式面临着另一个问题，即如何获得所需要的资源并在应用程序中使用资源。对此，可以采用两种方法。第一种办法是通过代码创建一个ResourceDictionary对象，再通过指定其Source属性来定位程序中资源字典文件，一旦创建了ResourceDictionary对象，就可以通过key来检索对应的资源，具体的实现代码如下：

这种方式不需要手动指定资源，当加载一个新的资源字典时，窗口中所有的DynamicResource引用都会自动引用新的资源，这样的方式可以用来构建动态的皮肤功能。

另外一种办法可以使用ComponentResourceKey.aspx)标记，使用ComponentResourceKey为资源创建键名。具体使用例子请参看博文：Defining and Using Shared Resources in a Custom Control Library。

三、WPF样式详解

在前面介绍了WPF资源，使用资源可以在一个地方定义对象而在整个应用程序中重用它们，除了在资源中可以定义各种对象外，还可以定义样式，从而达到样式的重用。

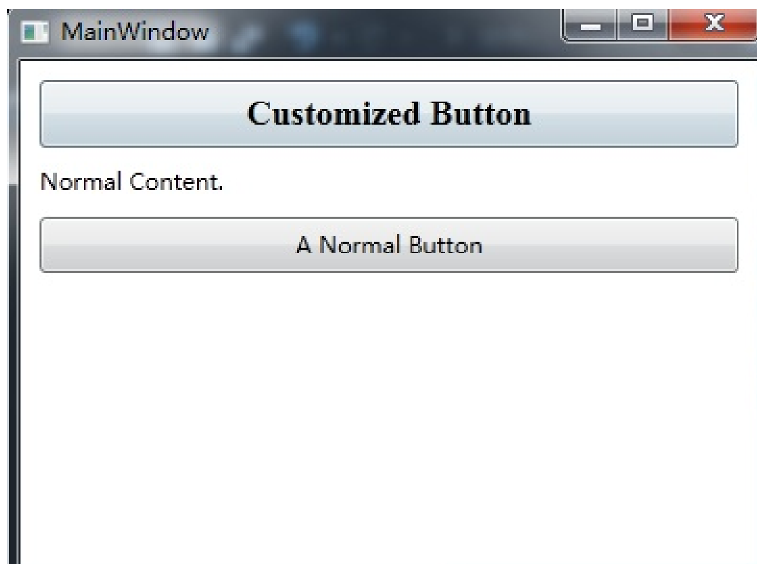
样式可以理解为元素的属性集合。与Web中的CSS类似。WPF可以指定具体的元素类型为目标，并且WPF样式还支持触发器，即当一个属性发生变化的时，触发器中的样式才会被应用。

3.1 WPF样式使用

之前WPF资源其实完全可以完成WPF样式的功能，只是WPF样式对资源中定义的对象进行了封装，使其存在于样式中，利于管理和应用，我们可以把一些公共的属性定义放在样式中进行定义，然后需要引用这些属性的控件只需要引用具体的样式即可，而不需要对这多个属性进行分别设置。下面XAML代码就是一个样式的使用示例：

```
<Window x:Class="StyleDemo.StyleDefineAndUse"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/present
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="300" Width="400">
    <Window.Resources>
        <!--定义样式-->
        <Style TargetType="Button">
            <Setter Property="FontFamily" Value="Times New Roman" /
            <Setter Property="FontSize" Value="18" />
            <Setter Property="FontWeight" Value="Bold" />
        </Style>
    </Window.Resources>
    <StackPanel Margin="5">
        <!--由于前面定义的样式没有定义key标记,如果没有显示指定Style为null,
        <Button Padding="5" Margin="5">Customized Button</Button>
        <TextBlock Margin="5">Normal Content.</TextBlock>
        <!--使其不引用事先定义的样式-->
        <Button Padding="5" Margin="5" Style="{x:Null}">A Normal Bu
    </StackPanel>
</Window>
```

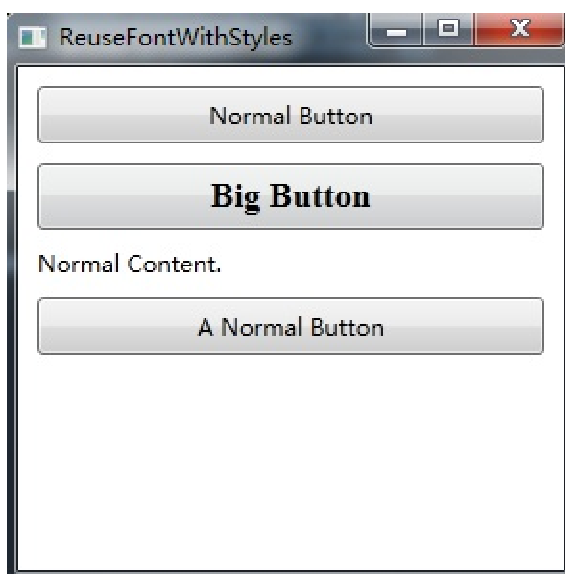
具体的运行效果如下图所示：



当样式中没有定义key标记时，则对应的样式会指定应用到目标对象上，上面XAML代码就是这种情况，如果显式为样式定义了key标记的话，则必须显式指定样式Key的方式，对应的样式才会被应用到目标对象上，下面具体看看这种情况。此时XAML代码如下所示：

```
<Window x:Class="StyleDemo.ReuseFontWithStyles"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/present
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ReuseFontWithStyles" Height="300" Width="300">
    <Window.Resources>
        <!--带有key标签的样式-->
        <Style TargetType="Button" x:Key="BigButtonStyle">
            <Setter Property="FontFamily" Value="Times New Roman" /
            <Setter Property="FontSize" Value="18" />
            <Setter Property="FontWeight" Value="Bold" />
        </Style>
    </Window.Resources>
    <StackPanel Margin="5">
        <!--如果不显式指定样式key将不会应用样式-->
        <Button Padding="5" Margin="5">Normal Button</Button>
        <Button Padding="5" Margin="5" Style="{StaticResource BigBu
        <TextBlock Margin="5">Normal Content.</TextBlock>
        <!--使其不引用事先定义的样式-->
        <Button Padding="5" Margin="5" Style="{x:Null}">A Normal Bu
    </StackPanel>
</Window>
```

此时运行效果如下图所示：



3.2 样式触发器

WPF样式还支持触发器，在样式中定义的触发器，只有在该属性或事件发生时才会被触发，下面具体看看简单的样式触发器是如何定义和使用的，具体的XAML代码如下所示：

```
<Window x:Class="StyleDemo.SimpleTriggers"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/present
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="SimpleTriggers" Height="300" Width="300">
    <Window.Resources>
        <Style x:Key="BigFontButton">
            <Style.Setters>
                <Setter Property="Control.FontFamily" Value="Times
                <Setter Property="Control.FontSize" Value="18" />

            </Style.Setters>
            <!--样式触发器-->
            <Style.Triggers>
                <!--获得焦点时触发-->
                <Trigger Property="Control.IsFocused" Value="True">
                    <Setter Property="Control.Foreground" Value="Re
                </Trigger>
                <!--鼠标移过时触发-->
                <Trigger Property="Control.IsMouseOver" Value="True
                    <Setter Property="Control.Foreground" Value="Ye
                    <Setter Property="Control.FontWeight" Value="Bo
                </Trigger>
                <!--按钮按下时触发-->
                <Trigger Property="Button.IsPressed" Value="True">
                    <Setter Property="Control.Foreground" Value="Bl
                </Trigger>
            </Style.Triggers>
        </Style>
    </Window.Resources>

    <StackPanel Margin="5">
        <Button Padding="5" Margin="5"
            Style="{StaticResource BigFontButton}"
              >A Big Button</Button>
        <TextBlock Margin="5">Normal Content.</TextBlock>
        <Button Padding="5" Margin="5"
            >A Normal Button</Button>
    </StackPanel>
</Window>
```

此时的运行效果如下图所示：

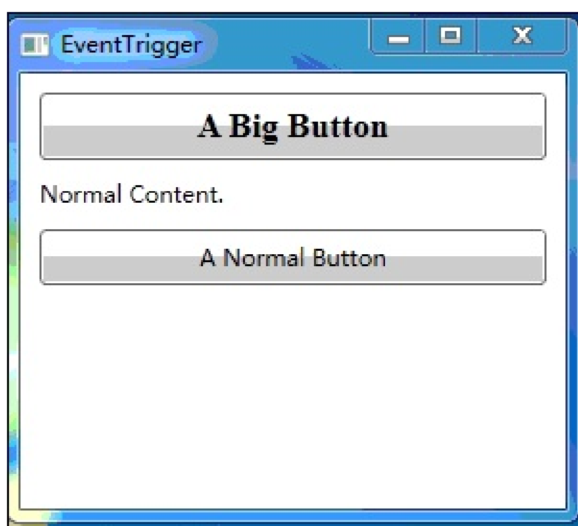


上面定义的触发器都是在某个属性发生变化时触发的，也可以定义当某个事件激活时的触发器，我们也把这样的触发器称为事件触发器，下面示例定义的事件触发器是等待MouseEnter.aspx)事件，一旦触发MouseEnter事件，则动态改变按钮的FontSize属性来形成动画效果，具体的XAML代码如下所示：

```
<Window x:Class="StyleDemo.EventTrigger"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/present
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="EventTrigger" Height="300" Width="300">
    <Window.Resources>
        <Style x:Key="BigFontButton">
            <Style.Setters>
                <Setter Property="Control.FontFamily" Value="Times
                <Setter Property="Control.FontSize" Value="18" />
                <Setter Property="Control.FontWeight" Value="Bold"
            </Style.Setters>
            <Style.Triggers>
                <!--定义事件触发器-->
                <EventTrigger RoutedEvent="Mouse.MouseEnter">
                    <!--事件触发时只需的操作-->
                    <EventTrigger.Actions>
                        <!--把动画放在动画面板中-->
                        <BeginStoryboard>
                            <!--在0.2秒的时间内将字体放大到22单位-->
                            <Storyboard>
                                <DoubleAnimation
                  Duration="0:0:0.2"
                  Storyboard.TargetProperty="FontSize"
                  To="22"  />
                            </Storyboard>
                        </BeginStoryboard>
                    </EventTrigger.Actions>
                </EventTrigger>
                <!--鼠标移开触发的事件-->
                <EventTrigger RoutedEvent="Mouse.MouseLeave">
                    <EventTrigger.Actions>
                        <BeginStoryboard>
                            <!--在1秒的时间内将字体尺寸缩小到原来的大小--
                            <!--如果目标字体尺寸没有明确指定，则WPF将默认
                            <Storyboard>
```

```
                                <DoubleAnimation
                  Duration="0:0:1"
                  Storyboard.TargetProperty="FontSize"  />
                            </Storyboard>
                        </BeginStoryboard>
                    </EventTrigger.Actions>
                </EventTrigger>
            </Style.Triggers>
        </Style>
    </Window.Resources>
    <StackPanel Margin="5">
        <Button Padding="5" Margin="5"
            Style="{StaticResource BigFontButton}"
              >A Big Button</Button>
        <TextBlock Margin="5">Normal Content.</TextBlock>
        <Button Padding="5" Margin="5"
            >A Normal Button</Button>
    </StackPanel>
</Window>
```

此时的运行效果如下图所示：



四、小结

到这里，WPF资源和样式的内容就介绍结束。总结为，WPF样式类似CSS，可以将多个属性定义在一个样式中，而样式又存放在资源中，资源成了样式和对象的容器。另外WPF样式还支持触发器功能，本文中演示了属性触发器和事件触发器的使用。在接下来一篇博文中将介绍WPF模板。

本文所有源码：ResourceAndStyle.zip

WPF快速入门系列(7)——深入解析WPF模板

一、引言

模板从字面意思理解是“具有一定规格的样板"。在现实生活中，砖块都是方方正正的，那是因为制作砖块的模板是方方正正的，如果我们使模板为圆形的话，则制作出来的砖块就是圆形的，此时我们并不能说圆形的”砖块“不是砖块吧。因为形状只是它们的外观，其制作材料还是一样的。所以，模板可以理解为表现形式。WPF中的模板同样是表现形式的意思。

在WPF中包括三种模板：控件模板、数据模版和面板模板。它们都继承于FrameworkTemplate.aspx)基类，其继承层次结果如下图所示：

◢继承层次结构

System.Object
  System.Windows.Threading.DispatcherObject
    System.Windows.FrameworkTemplate
      System.Windows.Controls.ControlTemplate
      System.Windows.Controls.ItemsPanelTemplate
      System.Windows.DataTemplate

从上图可以发现，FrameworkTemplate确实有三个子类，它们正是WPF中支持的三种模板。对于控件模板，即控件外观外衣，可以通过修改控件模板来自定义控件的外观表现，例如，可以通过修改按钮的控件模板使按钮表现为圆形；数据模板，即数据的外衣。用于从一个对象中提取数据，并在内容控件或列表控件的各个项中显示数据。面板模板即面板的外衣，而面板又用于进行布局的，所以面板的外衣也就是布局的外衣，通过修改面板模板可以自定义控件的布局。例如，ListBox默认是自从向下地显示每一项的，此时可以通过修改面板模板使其自左向右地显示每一项。

WPF模板其实都是外观的表现形式，不管是控件模板、数据模板还是面板模板，其都是改变控件的表现形式。只不过这三种控件的作用点不一样罢了。控件模板是针对于控件本身，修改它可以改变控件本身表现的样子；数据模板针对控件的数据，修改它可以改变控件绑定的数据表现样子。既然是决定数据的表现，从而决定其一般应用于数据绑定控件，如ListBox、ListView等控件。面板模板则针对于控件的布局，修改它可以影响控件的布局方式。

二、控件模板

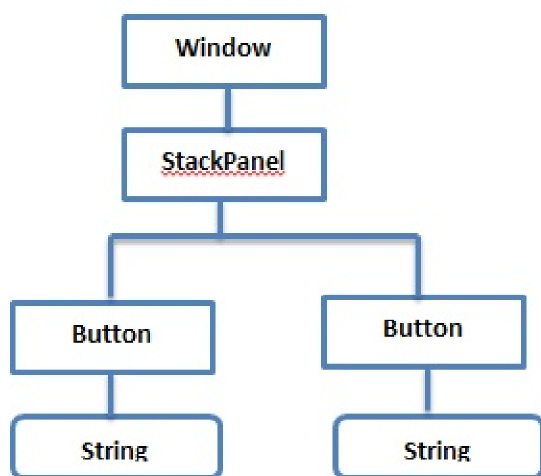
在分别介绍这三种控件模板之前，我觉得你有必要先了解WPF的逻辑树和可视化树的内容，因为你要修改控件模板，则首先需要了解控件的组成。

2.1 WPF的逻辑树和可视化树

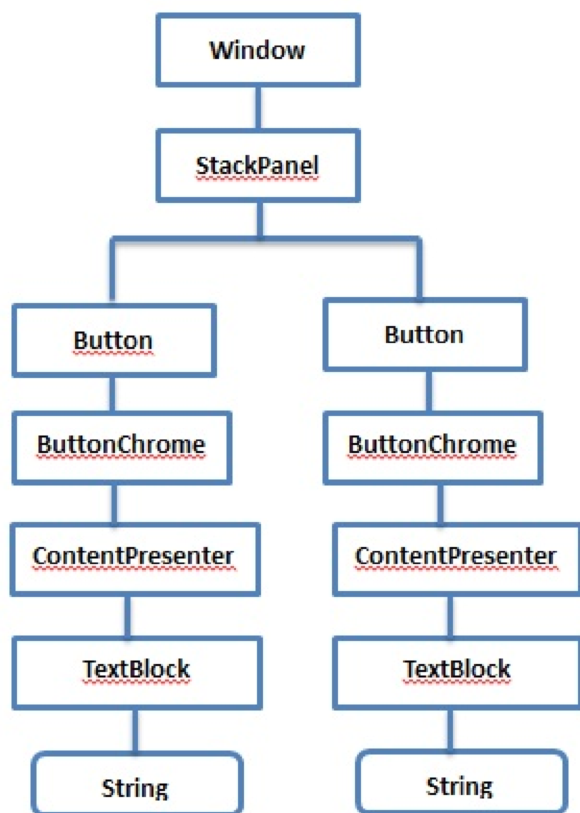
在许多技术中，元素和组件都是按树结构的形式进行组织的。使用这样的结构，开发人员可以直接操作树中的对象节点来程序对象，从而通过操作该对象来修改程序的表现和行为（这是了解逻辑树和可视化树的主要原因）。在WPF中，同样使用了树结构来组织元素之间的关系。WPF中支持逻辑树和可视化树的概念，并且WPF公开了两个提供树形视图帮助器类：LogicalTreeHelper.aspx) 和 VisualTreeHelper.aspx)。逻辑树指的是UI界面的组成元素的结构。先看下面的XAML代码的例子：

```
<Window x:Class="TemplateDemo.VisualTree"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/present
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="200" Width="300">
    <StackPanel Margin="5">
        <Button Padding="5" Margin="10">First Button</Button>
        <Button Padding="5" Margin="10">Second Button</Button>
    </StackPanel>
</Window>
```

上面XAML的逻辑树如下图所示：



可视化树是逻辑树的扩展版本，它将元素分成更小的部分。上面XAML代码对应的可视化树如下图所示：



从上面可视化树可以看出，Button由多个可视化元素组成——使按钮具有阴影背景特征的边框(由ButtonChrome.aspx)类表示)、内部的容器(一个ContentPresenter对象)以及存储按钮文本的文本块控件(由TextBlock表示)。上面的可视化树和逻辑树结构并不是我凭空想象出来的，而是有事实依据的，我们可以通过VisualTreeHelper类和LogicTreeHelper类提供的方法来查看窗口的可视化树和逻辑树，下面的例子实现了这个需求，具体的XAML实现如下所示：

```
<Window x:Class="TemplateDemo.Window1"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/present
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="380" Width="400">
    <StackPanel Margin="5">
        <Button Padding="5" Margin="5" Click="ShowLogicTree">Show L
        <Button Padding="5" Margin="5" Click="ShowVisualTree">Show
        <!--TreeView控件用来显示窗口的逻辑树和可视化树-->
        <TreeView Name="treeElements" Margin="5"></TreeView>
    </StackPanel>
</Window>
```

对应的后台代码实现入下所示：

```
public partial class Window1 : Window
    {
        public Window1()
        {
```

```
            InitializeComponent();
        }

        // 把公共代码抽象出一个方法，从而使代码重用
        public void ProcessElement(object obj, TreeViewItem item, T
        {
            item.Header = obj.GetType().Name;
            item.IsExpanded = true;

            // 如果当前元素是第一个元素就添加到树集合上
            // 如果是内嵌元素，则添加到它的父节点上
            if (previousItem == null)
            {
                treeElements.Items.Add(item);
            }
            else
            {
                previousItem.Items.Add(item);
            }
        }

        private void PrintLogicTree(object obj, TreeViewItem previo
        {
            TreeViewItem item = new TreeViewItem();
            ProcessElement(obj, item, previousItem);

            // 如果不是DependencyObject，则返回
            if (!(obj is DependencyObject))
                return;

            // 递归打印逻辑树
            foreach(object child in LogicalTreeHelper.GetChildren(o
            {
                // 这里为了避免死循环，因为TreeView的子元素包含Window1、S
                // 如果不加这个条件，控件会一直反复循环
                if (child is TreeView)
                    return;
                PrintLogicTree(child, item);
            }
        }

        private void PrintVisualTree(DependencyObject obj, TreeView
        {
            TreeViewItem item = new TreeViewItem();
            ProcessElement(obj, item, previousItem);

            //  递归输出视觉树
            for (int i = 0; i < VisualTreeHelper.GetChildrenCount(o
            {
                if (obj is TreeView)
                    return;

                PrintVisualTree(VisualTreeHelper.GetChild(obj, i),
```

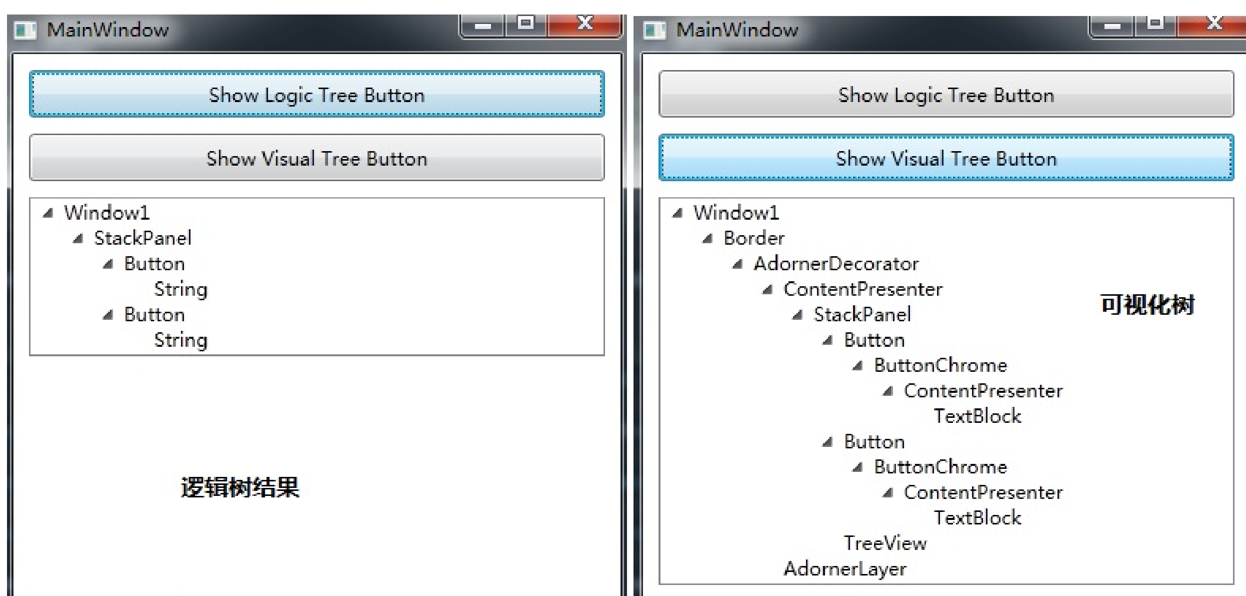
```
            }
        }

        private void ShowLogicTree(object sender, RoutedEventArgs e
        {
            treeElements.Items.Clear();
            PrintLogicTree(this, null);
        }

        private void ShowVisualTree(object sender, RoutedEventArgs
        {
            treeElements.Items.Clear();
            PrintVisualTree(this, null);
        }

    }
```

程序的运行效果如下图所示：

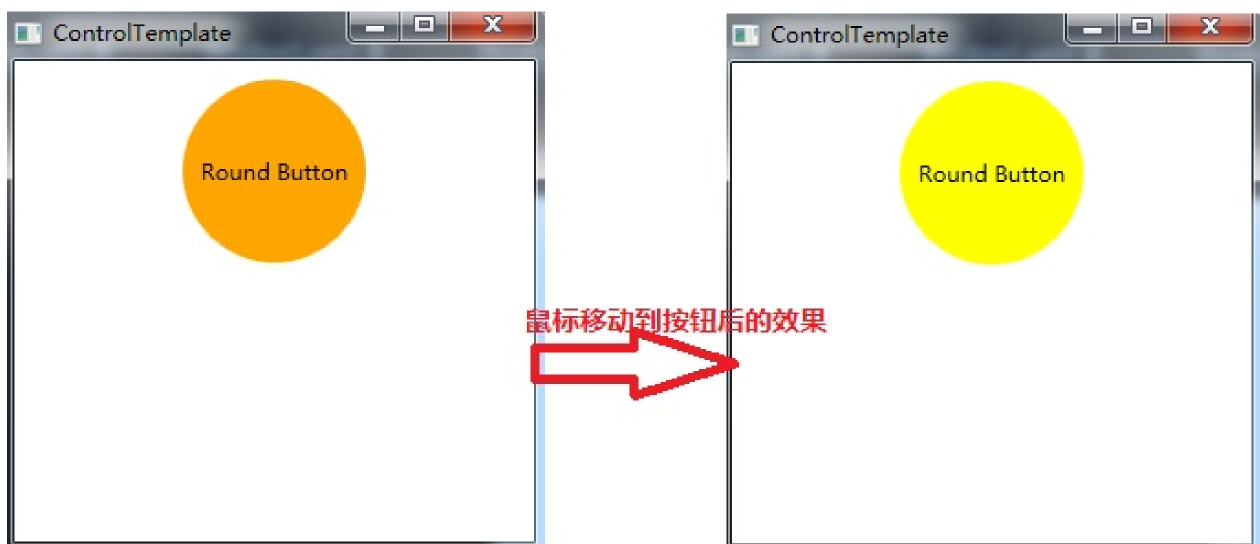


2.2 通过控件模板自定义控件外观

控件模板既然是控件的外衣，自然我们可以创建的新的控件模板，然后把新的控件模板应用到需要应用的控件中，这时候应用了新控件模板的控件，将会使用新的控件模板来渲染自身，从而改变控件的外观。这也是自定义控件外观的要旨。在WPF中按钮的默认控件是长方形的，我们可以通过创建一个新的控件模板来改变按钮的外观，下面的例子就实现了通过控件模板的方式自定义了一个圆形的按钮。具体的XAML代码如下所示：

```
<Window x:Class="TemplateDemo.ControlTemplate"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/present
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ControlTemplate" Height="300" Width="300">
    <Window.Resources>
        <!--定义控件模板，并使用key标记-->
        <ControlTemplate x:Key="roundButtonTemplate" TargetType="Bu
            <Grid>
                <Ellipse Name="ell" Fill="Orange" Width="100" Heigh
                <!--使用模板绑定来绑定按钮的内容-->
                <ContentPresenter Content="{TemplateBinding Button
            </Grid>
            <!--定义模板触发器-->
            <ControlTemplate.Triggers>
                <Trigger Property="IsMouseOver" Value="True">
                    <Setter TargetName="ell"  Property="Fill" Value
                </Trigger>
            </ControlTemplate.Triggers>
        </ControlTemplate>
    </Window.Resources>
    <StackPanel Margin="10">
        <Button Content="Round Button" Template="{StaticResource ro
    </StackPanel>
</Window>
```

此时，你就可以看到按钮是一个圆形的了，并且当鼠标移动到按钮上时，会触发模板触发器来改变Ellipse的填充色，具体的运行效果如下图所示：



从上面的控件模板的使用可知，它和创建自定义控件不同，在很多情况下，你不需要编写自己的控件，你只是希望更改控件的外观。使用控件面板非常简单：

- 首先在资源集合中创建一个**ControlTemplate**，并指定**key**标记
- 然后赋值到控件的**Template**属性中。

三、数据模板

数据模板是数据的外衣，数据模板是一段定义如何绑定数据对象的XAML标记，有两种类型的控件支持数据模板：

- 内容控件通过ContentTemplate属性支持数据模板。内容模板用于显示任何放在Content属性中的内容。
- 列表控件，即继承自ItemsControl类的控件，通过ItemPlate属性支持数据模板。该模板用于显示由ItemSource提供集合中的每一项。

基于列表的模板实际上是以内容控件模板为基础的，因为列表中的每一项由一个内容控件包装的。如ListBox控件的ListBoxItem元素。下面让我们具体看看如何去创建一个数据模板吧。

3.1 如何定义数据模板

具体的XAML代码如下所示：

```
<Window x:Class="TemplateDemo.DataTemplate"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/present
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local ="clr-namespace:TemplateDemo;assembly=TemplateD
        Title="DataTemplate" Height="300" Width="300">
    <Window.Resources>
        <!--创建数据模板-->
        <DataTemplate x:Key="personDataTem">
            <Border Name="blueBorder" Margin="3" BorderThickness="3
              CornerRadius="5">
                <Grid Margin="3">
                    <Grid.RowDefinitions>
                        <RowDefinition></RowDefinition>
                        <RowDefinition></RowDefinition>
                    </Grid.RowDefinitions>
                    <TextBlock Name="nametxt" FontWeight="Bold" Tex
                    <TextBlock Grid.Row="1" Text="{Binding Path=Age
                </Grid>
            </Border>
            <!--定义数据模板触发器-->
            <DataTemplate.Triggers>
                <Trigger SourceName="blueBorder" Property="IsMouseO
                    <Setter TargetName="blueBorder" Property="Backg
                    <Setter TargetName="nametxt" Property="FontSize
                </Trigger>
            </DataTemplate.Triggers>
        </DataTemplate>
    </Window.Resources>
    <StackPanel Margin="5">
        <ListBox Name="lstPerson" HorizontalContentAlignment="Stret
    </StackPanel>
</Window>
```

其对应的后台代码如下所示：

```
 public partial class DataTemplate : Window
    {
        ObservableCollection<Student> persons = new ObservableColle
        {
            new Student() { Name ="LearningHard", Age=25},
            new Student() { Name ="HelloWorld", Age=22}
        };
        public DataTemplate()
        {
            InitializeComponent();

            lstPerson.ItemsSource = persons;
        }
    }

public class Student : INotifyPropertyChanged
    {
        public string ID { get { return Guid.NewGuid().ToString();

        public string Name { get; set; }

        public int Age { get; set; }

        public event PropertyChangedEventHandler PropertyChanged;
        public void OnPropertyChanged(PropertyChangedEventArgs e)
        {
            if (PropertyChanged != null)
                PropertyChanged(this, e);
        }
    }
```

其运行效果如下图所示：



从上面数据模板的创建可知，使用DataTemplate很简单：

- 首先在资源集合中创建一个数据模板，并设置**key**标签。
- 然后将**key**赋值到控件的**CellTemplate**或**ContentTemplate**或**ItemTemplate**属性上即可。

3. 2 数据模板与控件模板的关系

从上面的介绍可知，控件只是数据和行为的载体，至于它本身长什么样子和数据长什么样子都是靠Template决定的。决定控件外观的是ControlTemplate，决定数据外观的是DataTemplate， 它们正是Control.aspx)类的Template和ContentTemplate两个属性的值。

一般来说，ControlTemplate内都有一个ContentPresenter， 这个ContentPresenter的ContentTemplate就是DataTemplate类型。所以数据模板和控件模板的关系如下图所示：



四、创建面板模板

ItemsPanelTemplate用于指定项的布局。 ItemsControl.aspx) 类型具有一个类型为ItemsPanelTemplate 的 ItemsPanel.aspx) 属性。

每种ItemsControl都有其默认的ItemsPanelTemplate。ListBox, the default uses the VirtualizingStackPanel.">对于 ListBox.aspx)，默认值使用 VirtualizingStackPanel.aspx)。 MenuItem, the default uses WrapPanel.">对于 MenuItem.aspx)，默认值使用 WrapPanel.aspx)。 StatusBar, the default uses DockPanel.">对于 StatusBar.aspx)，默认值使用 DockPanel.aspx)。

ListBox, the default uses the VirtualizingStackPanel.">MenuItem, the default uses WrapPanel.">StatusBar, the default uses DockPanel.">自定义面板模板与自定义数据面板和数据面板一样简单，一样只需要首先定义一个面板模板在资源集合中，然后将其Key指定给ItemsPanel属性即可。具体的XAML实现如下所示：

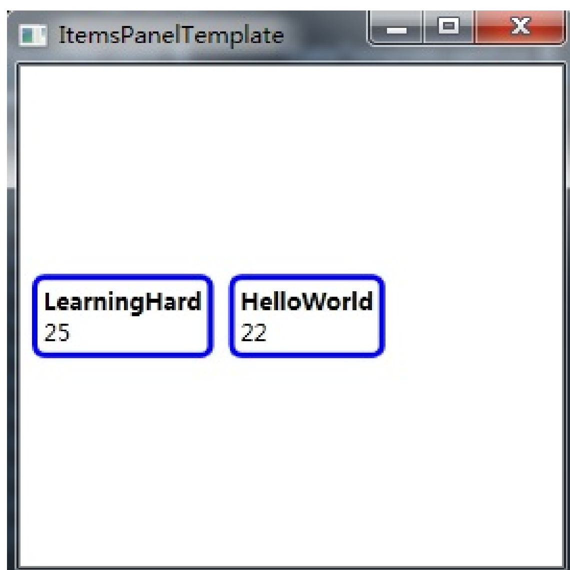
```
<Window x:Class="TemplateDemo.ItemsPanelTemplate"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/present
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ItemsPanelTemplate" Height="300" Width="300">
    <Window.Resources>
        <!--定义DataTemplate-->
        <DataTemplate x:Key="personDataTem">
            <Border Name="blueBorder" Margin="3" BorderThickness="3
              CornerRadius="5">
                <Grid Margin="3">
                    <Grid.RowDefinitions>
                        <RowDefinition></RowDefinition>
                        <RowDefinition></RowDefinition>
                    </Grid.RowDefinitions>
                    <TextBlock Name="nametxt" FontWeight="Bold" Tex
                    <TextBlock Grid.Row="1" Text="{Binding Path=Age
                </Grid>
            </Border>
        </DataTemplate>
        <!--定义ItemsPanelTemplate-->
        <ItemsPanelTemplate x:Key="listItemsPanelTem">
            <StackPanel Orientation="Horizontal"
                    VerticalAlignment="Center"
                    HorizontalAlignment="Left"/>
        </ItemsPanelTemplate>
    </Window.Resources>

    <!--使用ItemsPanelTemplate只需要赋值给ItemsPanel属性即可-->
    <ListBox Name="lstPerson" ItemsPanel="{StaticResource listItems
</Window>
```

其后台代码和数据模板的后台代码一样，其实现代码为：

```
public partial class ItemsPanelTemplate : Window
    {
        ObservableCollection<Student> persons = new ObservableColle
        {
            new Student() { Name ="LearningHard", Age=25},
            new Student() { Name ="HelloWorld", Age=22}
        };
        public ItemsPanelTemplate()
        {
            InitializeComponent();
            lstPerson.ItemsSource = persons;
        }
    }
```

此时程序运行的效果如下图所示，从下图结果可以看出，此时ListBox中的项不再是自上而下排列了，而是从左向右排列的。



五、总结

到这里，WPF模板的内容就介绍结束了，本文主要介绍了WPF中支持的三种模板：控件模板、数据模板和面板模板，然后各自定义并使用了自定义的模板，最后介绍了这三个模板之间的联系。使用这三个模板的方式都非常简单，都是先定义一个模板，然后在把对应的**key**应用到控件对应的属性中，对于控件模板，应用的是控件的**Template**，对于数据模板，应用的是控件的**ItemTemplate**属性，对于面板模板，应用的是控件的**ItemsPanel**属性。在下面的一篇博文将介绍如何实现一个MVVM的实例程序。

本文所有源码下载：TemplateDemo.zip

WPF快速入门系列(8)——MVVM快速入门

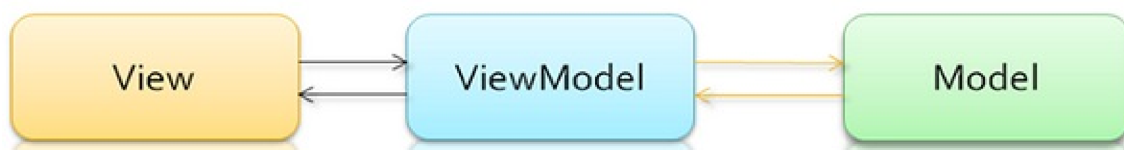
一、引言

在前面介绍了WPF一些核心的内容，其中包括WPF布局、依赖属性、路由事件、绑定、命令、资源样式和模板。然而，在WPF还衍生出了一种很好的编程框架，即WVVM，在Web端开发有MVC，在WPF客户端开发中有MVVM，其中VM就相当于MVC中C(Control)。在Web端，微软开发了Asp.net MVC这样的MVC框架，同样在WPF领域，微软也开发了Prism这样的MVVM框架。Prism项目地址是：http://compositewpf.codeplex.com/SourceControl/latest。大家有兴趣的可以下载源码研究下。

本文所有源码下载：FristMVVMProject.zip

二、MVVM模式是什么？

既然讲到MVVM模式，自然第一个问题就是MVVM的含义。MVVM是Model-View-ViewModel的缩写形式，它通常被用于WPF或Silverlight开发。这三者之间的关系如下图所示：



下面我们分别来介绍下这三部分。

模型(Model)

Model——可以理解为带有字段，属性的类。

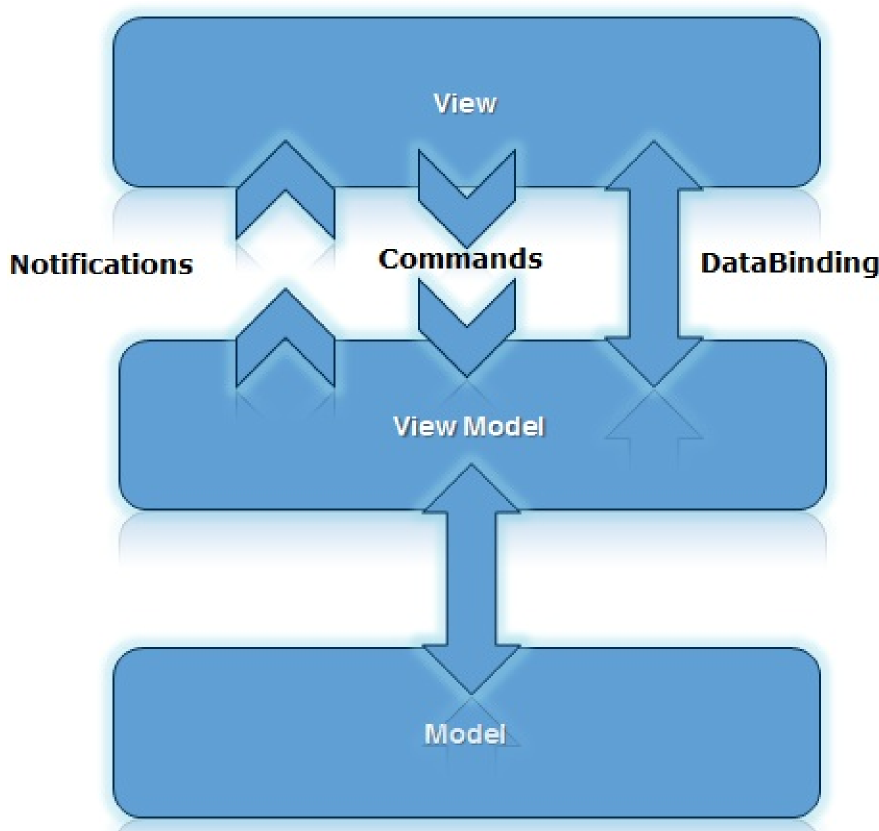
视图(View)

View——可以理解为我们所看到的UI。

视图模型(View Model)

View Model在View和Model之间，起到连接的作用，并且使得View和Model层分离。View Model不仅仅是Model的包装，它还包含了程序逻辑，以及Model扩展，例如，如果Model中有一个公开属性不需要在UI上显示，此时我们可以不再View Model中去定义它。

在MVVM模式下，WPF程序的运行流程如下图所示：



在MVVM中，VM的地位可以说是举足轻重。使用MVVM模式具有以下几个特点：

- 视图的cs文件包括极少的代码，其核心逻辑都被放在View Model类中，从而使得程序逻辑与视图耦合度降低。
- ViewModel类作为View的DataContext。
- 在MVVM下，所有的事件和动作都被当成命令，如按钮的点击操作，此时不是触发点击事件，而是绑定到一个点击命令，再由命令去执行对应的逻辑。

三、使用MVVM模式来实现WPF程序

前面介绍了MVVM一些基础知识，下面通过一个实例来说明下如何在WPF程序中应用MVVM模式。在之前实现WPF程序时，我们可能会把所有的后台逻辑都放在视图的后台文件中，这样的实现方式的好处更直观，方便，对于一些小的应用程序这样做当然没什么问题，但是对于复杂的应用程序这样写的话，可能会导致后台代码显得非常臃肿，到最好变得难以维护。此时想到的解决方案就是职责分离，使后台的逻辑分离到其他类中，MVVM其实我理解就是达到这个目的。下面我们按照MVVM的组成部分来实现下这个MVVM程序。

第一步：自然是数据部分了，即**Model**层的实现。在这里定义了一个**Person**类，其中包含了**2**个基本的属性。

为了进行测试，下面创建一个静态方法来获得测试数据。

```
public class PersonDataHelper
    {
        public static ObservableCollection<Person> GetPersons()
        {
            ObservableCollection<Person> samplePersons = new Observ
            samplePersons.Add(new Person() {Name = "张三", Age = 33
            samplePersons.Add(new Person() { Name ="王五", Age= 22
            samplePersons.Add(new Person() { Name = "李四", Age = 3
            samplePersons.Add(new Person() { Name = "LearningHard",
            return samplePersons;
        }
    }
```

第二步：实现**ViewModel**层，实现数据和界面之间的逻辑。在视图模型类中，包含了属性和命令，因为在MVVM中，事件都当成命令来进行处理，其中命令只能与具有Command属性的控件进行绑定。既然要包含命令，首先就需要实现一个命令，这里自定义的命令需要实现ICommand接口。这里我们定义了一个QueryCommand。具体的实现代码如下所示：

```
public class QueryCommand :ICommand
    {
        #region Fields
        private Action _execute;
        private Func<bool> _canExecute;
        #endregion

        public QueryCommand(Action execute)
            : this(execute, null)
        {
        }
        public QueryCommand(Action execute, Func<bool> canExecute)
        {
            if (execute == null)
                throw new ArgumentNullException("execute");
            _execute = execute;
            _canExecute = canExecute;
        }

        #region ICommand Member

        public event EventHandler CanExecuteChanged
        {
            add
            {
                if (_canExecute != null)
                {
                    CommandManager.RequerySuggested += value;

                }
```

```
            }
            remove
            {
                if (_canExecute != null)
                {
                    CommandManager.RequerySuggested -= value;

                }
            }
        }

        public bool CanExecute(object parameter)
        {
            return _canExecute == null ? true : _canExecute();
        }

        public void Execute(object parameter)
        {
            _execute();
        }
        #endregion
    }
```

接下来就是定义我们的ViewModel类了，具体的实现代码如下所示：

```
 1 public class PersonListViewModel : INotifyPropertyChanged
 2     {
 3         #region Fields
 4         private string _searchText;
 5         private ObservableCollection<Person> _resultList;
 6         #endregion
 7 
 8         #region Properties
 9 
10         public ObservableCollection<Person> PersonList { get; pr
11 
12         // 查询关键字
13         public string SearchText
14         {
15             get { return _searchText; }
16             set
17             {
18                 _searchText = value;
19                 RaisePropertyChanged("SearchText");
20             }
21         }
22 
23         // 查询结果
24         public ObservableCollection<Person> ResultList
25         {
```

```
26              get { return _resultList; }
27              set
28              {
29                  _resultList = value;
30                  RaisePropertyChanged("ResultList");
31              }
32          }
33
34          public ICommand QueryCommand
35          {
36              get { return new QueryCommand(Searching, CanSearchin
37          }
38
39          #endregion
40
41          #region Construction
42          public PersonListViewModel()
43          {
44              PersonList = PersonDataHelper.GetPersons();
45              _resultList = PersonList;
46          }
47
48          #endregion
49
50          #region Command Handler
51          public void Searching()
52          {
53              ObservableCollection<Person> personList = null;
54              if (string.IsNullOrWhiteSpace(SearchText))
55              {
56                  ResultList = PersonList;
57              }
58              else
59              {
60                  personList = new ObservableCollection<Person>();
61                  foreach (Person p in PersonList)
62                  {
63                      if (p.Name.Contains(SearchText))
64                      {
65                          personList.Add(p);
66                      }
67                  }
68                  if (personList != null)
69                  {
70                      ResultList = personList;
71                  }
72              }
73          }
74
75          public bool CanSearching()
76          {
77              return true;
78          }
```

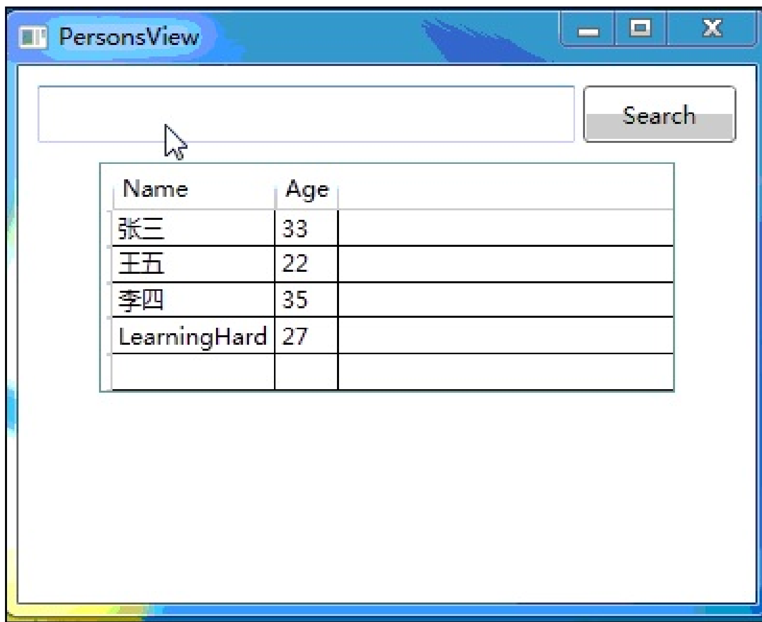
```
79
80          #endregion
81
82          #region INotifyPropertyChanged Members
83
84          public event PropertyChangedEventHandler PropertyChanged
85
86          #endregion
87
88          #region Methods
89          private void RaisePropertyChanged(string propertyName)
90          {
91              // take a copy to prevent thread issues
92              PropertyChangedEventHandler handler = PropertyChange
93              if (handler != null)
94              {
95                  handler(this, new PropertyChangedEventArgs(prope
96              }
97          }
98          #endregion
99      }
```

第三步：实现**View**层，设计我们的视图，设置它的**DataContext**属性为**ViewModel**类。具体的XAML代码如下所示：

```
<Window x:Class="MVVMDemo.View.PersonsView"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/present
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:MVVMDemo.ViewModel"
        Title="PersonsView" Height="350" Width="400">
    <!--设置DataContex是ViewModel类，当然你也可以使用后台代码设置-->
    <Window.DataContext>
        <local:PersonListViewModel />
    </Window.DataContext>
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="50"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>
        <TextBox Grid.Row="0" Name="searchtxt" Text="{Binding  Path
        <Button Grid.Row="0" Name="searchBtn" Content="Search" Comm
        <DataGrid Grid.Row="1" Name="datGrid"
                  HorizontalAlignment="Center"
                  VerticalAlignment="Top" ItemsSource="{Binding Pat

    </Grid>
</Window>
```

到此，我们的MVVM的WPF程序就已经完成了，下面就是要看看程序是否达到我们预期的目的。具体的运行结果如下图所示：



四、总结

到这里，本文的内容就分享完了，并且本文也是WPF系列的最后一篇了，希望这个系列可以使得初学者快速上手WPF编程。在接下来的时间里，我打算写一些具有实战性的内容，因为我之前都是分享一些初级的入门系列，接下来打算分享一些实际的项目实现，以及领域驱动设计方面的内容，希望得到大家的督促和支持。

WPF快速入门系列(9)——WPF任务管理工具实现

转载自：http://www.cnblogs.com/shanlin/p/3954531.html

WPF系列自然需要以一个实际项目为结束。这里分享一个博客园博客实现的一个项目，我觉得作为一个练手的项目非常合适。担心博主后期会删除什么，这里先备份在自己的博客里面分享给大家。

本文所有源码下载：TaskScheduler.zip

时光如梭，距离第一次写的 WPF学习开发客户端软件-任务助手(已上传源码) 已有三个多月，期间我断断续续地对该项目做了优化、完善等等工作，现在重新向大家介绍一下，希望各位可以使用，本软件以实用性为主，采用MVVM模式（有小部分没有修改过来），小巧、使用方便。

具体功能与更新如下：

计划助手:本软件由m.sh.lin0328@163.com开发与维护，免费使用，如有好的意见或建议，可发送邮件到m.sh.lin0328@163.com，谢谢使用！ 注（功能与特色）：1.本软件使用方便、操作简便； 2.本软件可设置任务运行周期：一次、每月、每周、每天、每小时、间隔分钟一共6种模式，满足您的不同需求； 2.本软件有定时运行任务（支持参数）、定时提醒、定时关机、定时关闭/打开显示器、定时锁屏、记事、天气预报等功能； 3.本软件声音文件在安装目录下的Audio文件夹下，拷贝进去即可（支持.mp3、.wma、.wmv等）； 4.增加最新资讯信息；

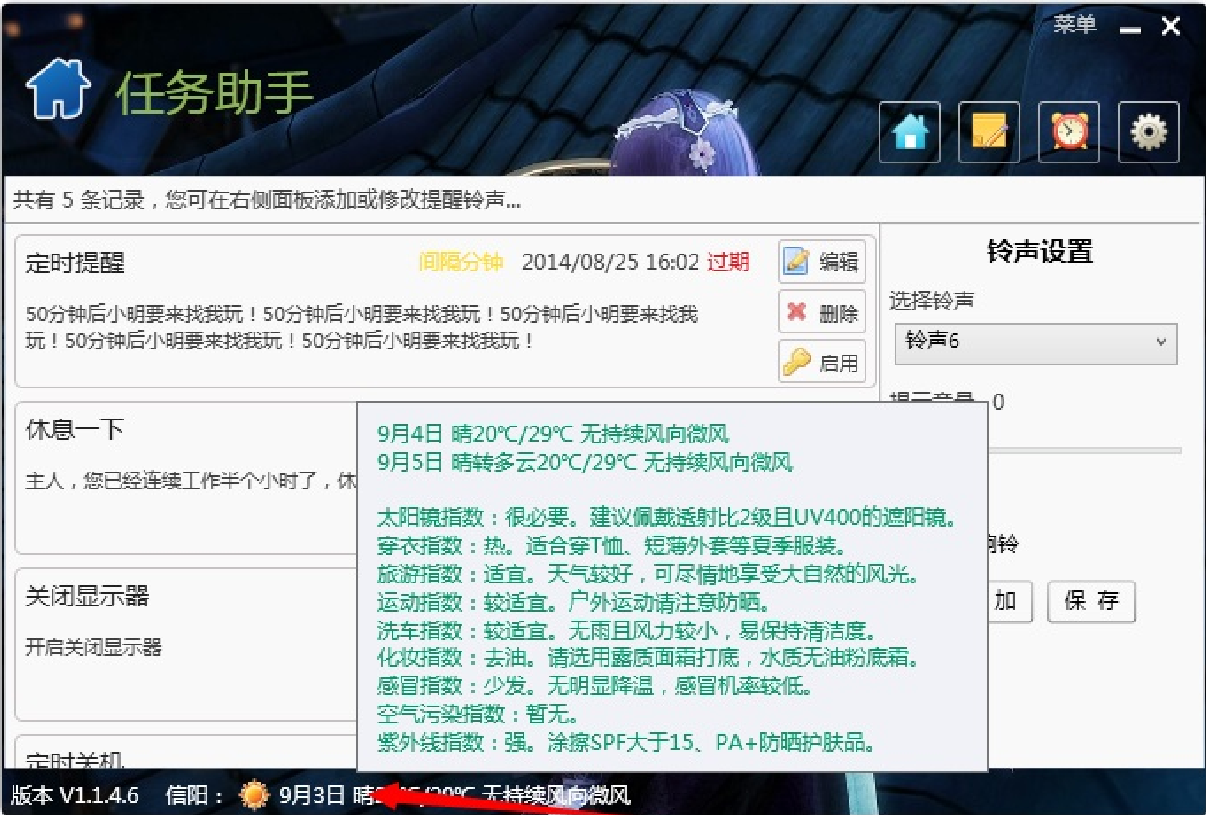
版本更新说明如下： 01.v.1.0.0.0 :2014-04-16：基本完成编码，添加快捷键 02.v.1.1.0.0 :2014-04-17: 增加开机启动，界面、托盘图标调整 03.v.1.1.2.0 :2014-05-01: 托盘修改 04.v.1.1.2.6 :2014-05-03: 窗体样式修改、提示声音修改 05.v.1.1.3.2 :2014-05-10: 主窗体列表样式修改，增加打开显示器等其它功能和细节 06.v.1.1.3.4 :2014-05-11: 任务详细窗体样式修改，增加过期和失效状态，解决关闭右下角提示不能关闭声音和其它细节 07.v.1.1.3.5 :2014-05-17: 任务状态增加失效与过期，增加锁屏功能，增加设置窗体，程序启动温馨提示功能 08.v.1.1.4.2 :2014-05-24: 数据存储改为SQLite,去除底栏状态，增加记事功能、铃声详细设置、增加天气预报、首页统计图表等及其它细节 09.v.1.1.4.4 :2014-08-23: 修复SQLite自启动报错，去除首页统计图表，任务运行周期增加按周运行，界面布局样式调整，记事增加翻页功能 10.v.1.1.4.6 :2014-08-30: 增加最新信息资讯功能，修复天气预报地域显示速度

新版本规划： 提醒功能增加推迟、声音重复播放次数、完善桌面提醒与节日提醒功能等

ExplorerMan

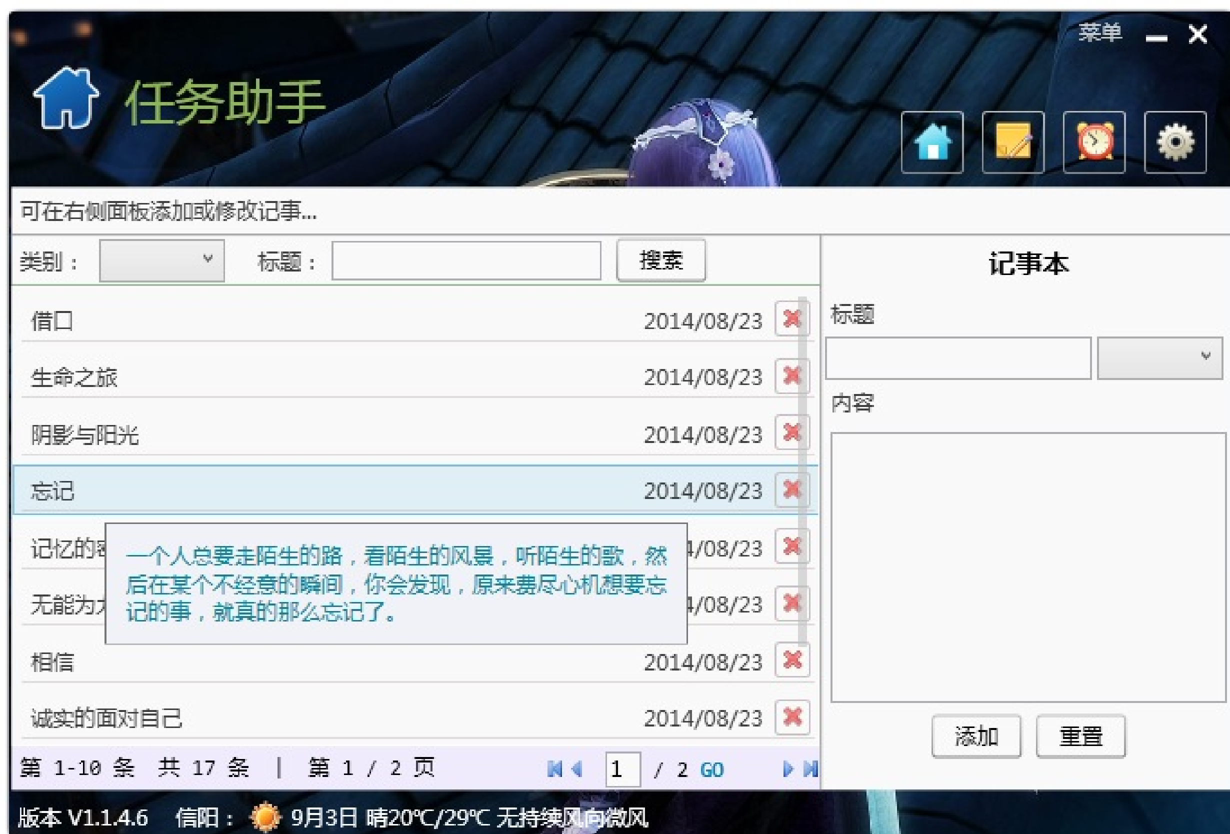
Audio
Bg
Down
Log
News
Weather
config.xml
GalaSoft.MvvmLight.Extras.WPF4.dll
GalaSoft.MvvmLight.WPF4.dll
Microsoft.Practices.ServiceLocation.dll
MSL.Tool.dll
System.Data.SQLite.dll
System.Windows.Interactivity.dll
Task.db
TimedTask.exe
TimedTask.exe.config
说明.txt

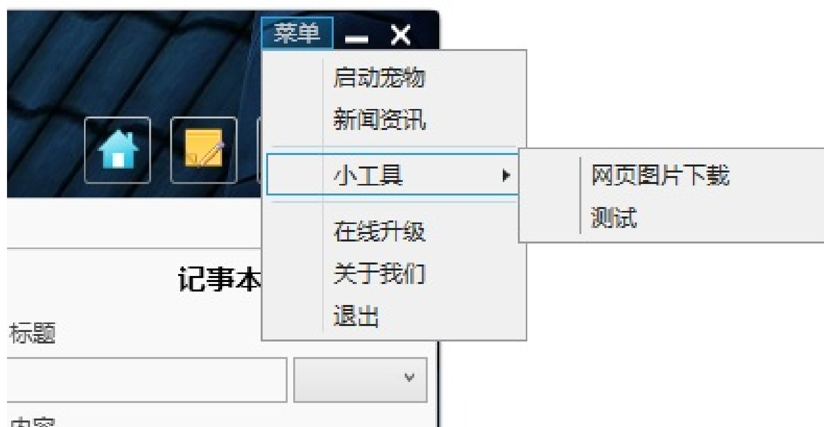



菜单
任务助手
共有 5 条记录，您可在右侧面板添加或修改提醒铃声...
定时提醒
间隔分钟 2014/08/25 16:02 过期
编辑
删除
启用
50分钟后小明要来找我玩！50分钟后小明要来找我玩！50分钟后小明要来找我玩！50分钟后小明要来找我玩！50分钟后小明要来找我玩！
铃声设置
选择铃声
铃声6
休息一下
主人，您已经连续工作半个小时了，休
关闭显示器
开启关闭显示器
9月4日 晴20℃/29℃ 无持续风向微风
9月5日 晴转多云20℃/29℃ 无持续风向微风
太阳镜指数：很必要。建议佩戴透射比2级且UV400的遮阳镜。
穿衣指数：热。适合穿T恤、短薄外套等夏季服装。
旅游指数：适宜。天气较好，可尽情地享受大自然的风光。
运动指数：较适宜。户外运动请注意防晒。
洗车指数：较适宜。无雨且风力较小，易保持清洁度。
化妆指数：去油。请选用露质面霜打底，水质无油粉底霜。
感冒指数：少发。无明显降温，感冒机率较低。
空气污染指数：暂无。
紫外线指数：强。涂擦SPF大于15、PA+防晒护肤品。
加
保 存
版本 V1.1.4.6
信阳：
9月3日 晴


温馨提示！
农历：农历二零一四年八月初十
时辰：庚戌
属相：马
星座：处女座


计划任务配置
基本设置
任务名称
定时任务
详细路径
浏览
启动参数
无
启用
备 注
任务列表
定时提醒
休息一下
关闭显示器
定时关机
温馨提醒
周期设置
一次
每月
每周
每天
每小时
间隔分钟
星期日
星期一
星期二
星期三
星期四
星期五
星期六
启动时间
2014/9/3
20
时
39
分
停止时间
2014/9/13
20
时
39
分
添加任务
重 置
状态：


菜单
任务助手
可在右侧面板添加或修改记事...
类别：
标题：
搜索
记事本
借口 2014/08/23
生命之旅 2014/08/23
阴影与阳光 2014/08/23
忘记 2014/08/23
相信 2014/08/23
诚实的面对自己 2014/08/23
一个人总要走陌生的路，看陌生的风景，听陌生的歌，然后在某个不经意的瞬间，你会发现，原来费尽心机想要忘记的事，就真的那么忘记了。
标题
内容
添加
重置
第 1-10 条 共 17 条 | 第 1 / 2 页
1 / 2 GO
版本 V1.1.4.6 信阳： 9月3日 晴20℃/29℃ 无持续风向微风


菜单
启动宠物
新闻资讯
小工具
在线升级
关于我们
退出
网页图片下载
测试
记事本
标题


任务助手 - 新闻资讯
国际
国内
健康
焦点
军事
体育
娱乐
我的导航
美国犹他州正式宣布支持一夫多妻制(图)
慰安妇陈亚扁：日军的蹂躏曾让她六度流产(图)
男子与岳父成为情人 自杀未果枪杀对方(图)
组图：镜头下美国少女真实生活状态
日本狗狗偷戴老奶奶假牙露着牙龈 萌翻网友
俄生物卫星返回地球 太空乘客壁虎全"殉职"(图)
英少女为庆祝成年 让15人舔其乳房上奶油(图)
日本G奶女星18岁陪酒 自曝完美脸蛋全靠整(图)
超模网晒"小狗荼胸"照 小狗次日被送医(图)
墨西哥长寿老妇127岁 一口气能睡3天(图)
全球要闻速递：美主持人称希拉里会成下任总统
世界经济论坛报告称中国全球竞争力稳中有升
09-03
优酷
网易
微信
百度贴吧
博客园
京东
天猫
招商银行

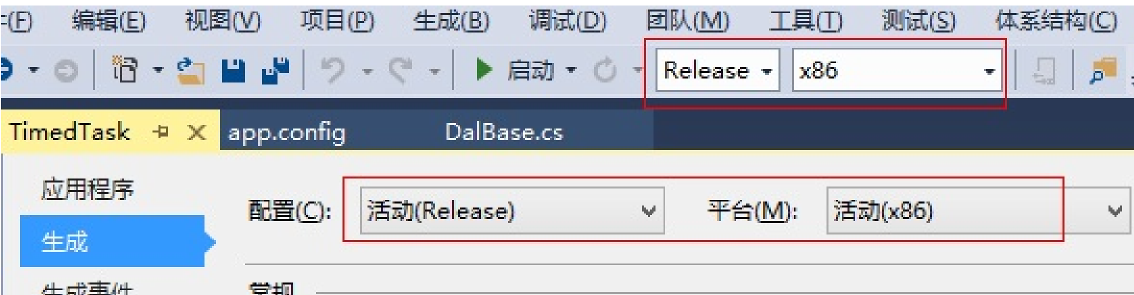

任务助手
版 本 号：V1.1.4.6
版权所有：m.sh.lin0328@163.com
Copyright@ 2014 Lin.All Right Reserved.
作品说明：本作品只用于学习交流, 如若转载, 请注明作者与出处。
http://home.cnblogs.com/u/shanlin/
查看使用说明
确 定


任务助手 - 设置中心
常规设置
快捷设置
常规设置
开机启动任务助手
点击主界面关闭按钮时最小化到系统托盘
记录运行日志
启动后显示资讯
系统背景
浏览
锁屏背景
浏览
天气预报
河南
信阳
锁屏时间
5
焦作
鹤壁
新乡
安阳
濮阳
许昌
漯河
三门峡
南阳
商丘
信阳
周口
驻马店
保存


主人您已经连续工作了1个小时，请注意休息！
距解锁剩余 10 秒！


隐藏窗口
总在最前
退出
自定义...
desk.chinaz.com

点击下载


编辑(E)
视图(V)
项目(P)
生成(B)
调试(D)
团队(M)
工具(T)
测试(S)
体系结构(C)
启动
Release
x86
TimedTask
app.config
DalBase.cs
应用程序
生成
配置(C):
活动(Release)
平台(M):
活动(x86)

ASP.NET 开发

ASP.NET 开发必备知识点(1)：如何让Asp.net网站运行在自定义的Web服务器上

一、前言

大家都知道，在之前，我们Asp.net 的网站都只能部署在IIS上，并且IIS也只存在于Windows上，这样Asp.net开发的网站就难以做到跨平台。由于微软的各项技术的开源，所以微软自然要对跨平台做出支持的。OWIN技术就可以使得Web 服务器不再依赖于IIS，从而使得Asp.net 网站不再依赖于Windows。是不是有了OWIN，就不需要安装MONO就可以实现跨平台呢？显然不是，有了OWIN要实现跨平台还是要依赖与MONO，因为MONO提供了在Liunx环境下.NET代码的运行环境，而OWIN只是分离了Web应用程序与Web Server之间的紧耦合罢了。

二、使Asp.net网站跨平台成为可能的机制——OWIN

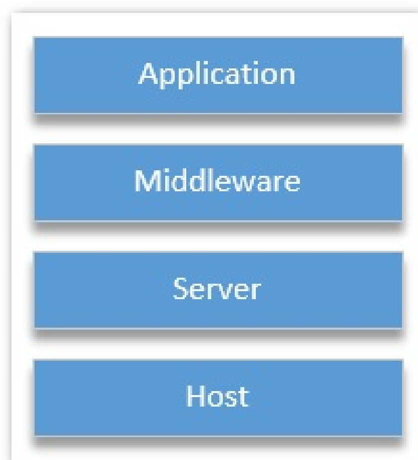
前面我们已经引出了使得Asp.net网站跨平台成为可能的机制就是OWIN，下面让我们具体看看什么是OWIN。

OWIN全称是——Open Web Interface For .NET。从名字上可以看出，它是一套接口定义，它完整定义如下：

OWIN在.NET Web Servers与Web Application之间定义了一套标准接口，OWIN的目标是用于解耦**Web Server**和**Web Application**。基于此标准，鼓励开发者开发简单、灵活的模块，从而推进.NET Web Development开源生态系统的发展。

至于为什么需要OWIN，在前面部分已经介绍过了，就是为了使得Web Application和Web Server解耦，这样就可以使得Asp.net 网站不再依赖与IIS Web Server，从而就不会紧耦合与Windows 操作系统了。（看到这里，你是不是和我学习OWIN有一样的疑问呢？问题是：之前没有OWIN规范不是照样可以通过Mono来实现asp.net 网站的跨平台吗？现在还需要OWIN干什么的？）

对于这个上面的疑问，我后面给出答案。既然OWIN是一套规范，则自然有它定义规范了。OWIN规范中定义了4个组件：



Host：主要负责托管应用程序的进程，可以是IIS，也可以自己写的程序等。主要用来启动，加载OWIN组件，以及合理地管理它们。

Server：指的实际的Web Server，负责绑定套接字并对Http请求进行监听，将Request和ReponsedeBody、Header封装成服务OWIN规范的字典并发送到OWIN Middleware Pipeline中进行处理。

Middleware：这个中间件就是用来在OWIN管道中处理请求的组件（可以把它想象成一个自定义的**Http Module**），它会被注册到Owin管道中一起处理Http request。

Application：这个就是我们自己开发的应用程序，或者是网站。

应用程序代理（**Application Delegate**）

Owin规范另一个重要的组成部分是接口的定义，它通过将服务器与应用程序之间的交互归纳为一个方法签名，称之为“应用程序代理”（Applacation Delegate）。具体定义如下：

上面委托的定义中第一参数称为环境字典，而第二个参数Task指的异步执行的方法。之前我们通过HttpContext对象来获得request、Response等对象，基于Owin的应用是通过这个环境字典来获得相应的对象。有了Owin之后，我们就不再与Asp.net管道打交道了，取而代之则是Owin管道。

Microsoft对OWIN规范的实现——Katana

既然OWIN是一套规范，自然就有其具体的实现，微软根据OWIN规范在Windows下实现了Katana（武士刀）。其开源地址：http://katanaproject.codeplex.com/。

Katana实现了OWIN的4个组件。

1）Host: Kataba为我们提供了3种Host的选择：

- IIS:使用IIS是最简单和向后兼容方式。在这种场景中OWIN管道通过标准的HttpModule和HttpHandler启动。使用此Host你必须使用System.Web作为OWIN Server
- Custom Host: 你也可以选择创建一个自定义宿主来托管应用程序
- OwinHost:Katana自己实现了宿主程序——OwinHost.exe。我们可以利用该宿

主来宿主我们的应用程序。

2）Server: Katana对Owin Server的实现提供如下几类实现：

- System.Web: System.Web与IIS两者彼此耦合，当你选择使用System.Web作为Server,此时必须选择IIS为宿主。
- HttpListener:这是OwinHost.exe和定义Host默认的Server。
- WebListener:这是ASP.NET vNext默认的轻量级Server。它目前无法使用在Katana张。

3） Middleware: 中间件（Middleware）用来处理Pipeline中的请求。Middleware是Owin Pipeline中处理请求的单元，它可以是Log组件，也可以是Asp.net Web API、SignlR等组件。

4）Application：应用程序的实现代码，可以为Asp.net MVC站点，也可以为Asp.net Web API和SignalR具体的应用实现。

Katana，它只能够运行在Windows中，使得在Windows环境下，我们的Asp.net 网站不完全依赖于IIS；而在Liunx环境下也有OWIN规范的具体实现，就是Jexus Web Server(简称JWS)。所以，我们可以利用Mono+OWIN+Jexus在Liunx环境下部署我们的Asp.net站点，具体部署参考：ASP.NET Linux部署(2) - MS Owin + WebApi + Mono + Jexus。

到这里，你们还记得我文章开头的疑问吗？大家应该都知道，在OWIN规范出现之前，我们就已经可以利用Mono来讲我们的Asp.net 站点部署在Liunx环境下了，之前采用的部署方式是：Mono+Apache/nginx + XSP2。具体部署请参考：Linux下的.NET之旅：第一站，CentOS+Mono+Xsp构建最简单的ASP.NET服务器。既然以前也可以实现Asp.net 网站在liunx环境下部署，则利用Owin的实现Jexus自然就有其优势，不能也就没有其存在的意义了，这里就涉及到Mono Xsp与基于Owin实现Jexus的一个对比：

Mono Xsp 和*Jexus*有什么区别呢：

1. 速度方面*:* 对于*ASP.NET*网页，大压力访问时*Jexus*处理速度更快*;* 对于静态文件，*Jexus*远快于*XSP*，而且对磁盘的要求和影响小*N*倍；
2. 功能方面*: XSP*是以*ASP.NET*测试工作开发的，功能单调，而*Jexus*是作为生产环境使用的真实的*WEB*服务开发的，功能全面，因此，*xsp*与*Jexus*在功能上可比性
3. 稳定性方面*: Jexus*有良好的容错和自动纠错能力，可以长期不间断运行，而*XSP*是单进程程序，没有任何自动纠错机制，无法保持不间断运行。
4. 安全性方面*: Jexus*有关键的入侵检测功能，*XSP*没有任何安全检测功能，没有可比性；
5. 多站点支持*: XSP*支持一站，*Jexus*支持任意多网站。

更详细内容请参考：*http://www.cnblogs.com/alsw/p/3255984.html*。

三、使用IIS托管Katana-based Asp.net网站

因为Katana为了向后兼容，依然支持IIS作为宿主，下面通过一个例子看看如何将Asp.net 站点托管在Katana-based的IIS中。

1. 创建一个空的Web Application：

2. 从Nuget中添加 Microsoft.Owin.Host.SystemWeb包

3. 添加OWIN Startup类，并添加如下代码在Startup1.Configuration方法中：

```
public void Configuration(IAppBuilder app)
    {
        // 有关如何配置应用程序的详细信息，请访问 http://go.microsoft.com
        **app.Run(context =>
        {
            context.Response.ContentType = "text/plain";
            return context.Response.WriteAsync("Hello, world."****)
        });**
    }
```

按F5运行，你将看到浏览器中打印出“Hello, world”的字样。

虽然同样是托管在IIS，但是所有的请求都会被OWIN来处理。Kanata除了支持IIS托管外，还支持自定义宿主，接下来介绍就是通过创建一个控制台程序来宿主Web 应用程序。

四、利于**Microsoft.Owin.Host.HttpListener**实现自寄宿

OWIN目标就是使得Web Server与Web Application解耦，接下来就具体看看如何将Web应用程序实现自我宿主。

1. 首先创建一个控制台应用程序

2. 通过Nuget安装Microsoft.Owin.Hosting和Microsoft.Owin.HttpListener包

3. 创建OWIN Startup类，该类的具体实现代码:

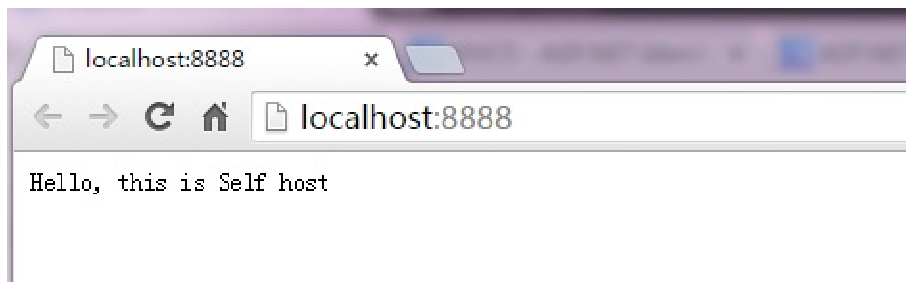
```
 public void Configuration(IAppBuilder app)
        {
            // 有关如何配置应用程序的详细信息，请访问 http://go.microsoft
            app.Run(context =>
            {
                context.Response.ContentType = "text/plain";
                return context.Response.WriteAsync("Hello, this is
            });
        }
```

4. 在Main方法中加入下面代码来启动我们的网站：

```
static void Main(string[] args)
        {
            using (WebApp.Start<Startup>(
            new StartOptions(url: "http://localhost:8888")))
            {
                Console.ReadLine();
            }

            Console.ReadLine();
        }
```

运行该控制台程序，然后在浏览器中输入“http://localhost:8888/”将看到如下界面：



当然，Katana还支持OwinHost.exe程序来进行宿主，其实现步骤如下所示：

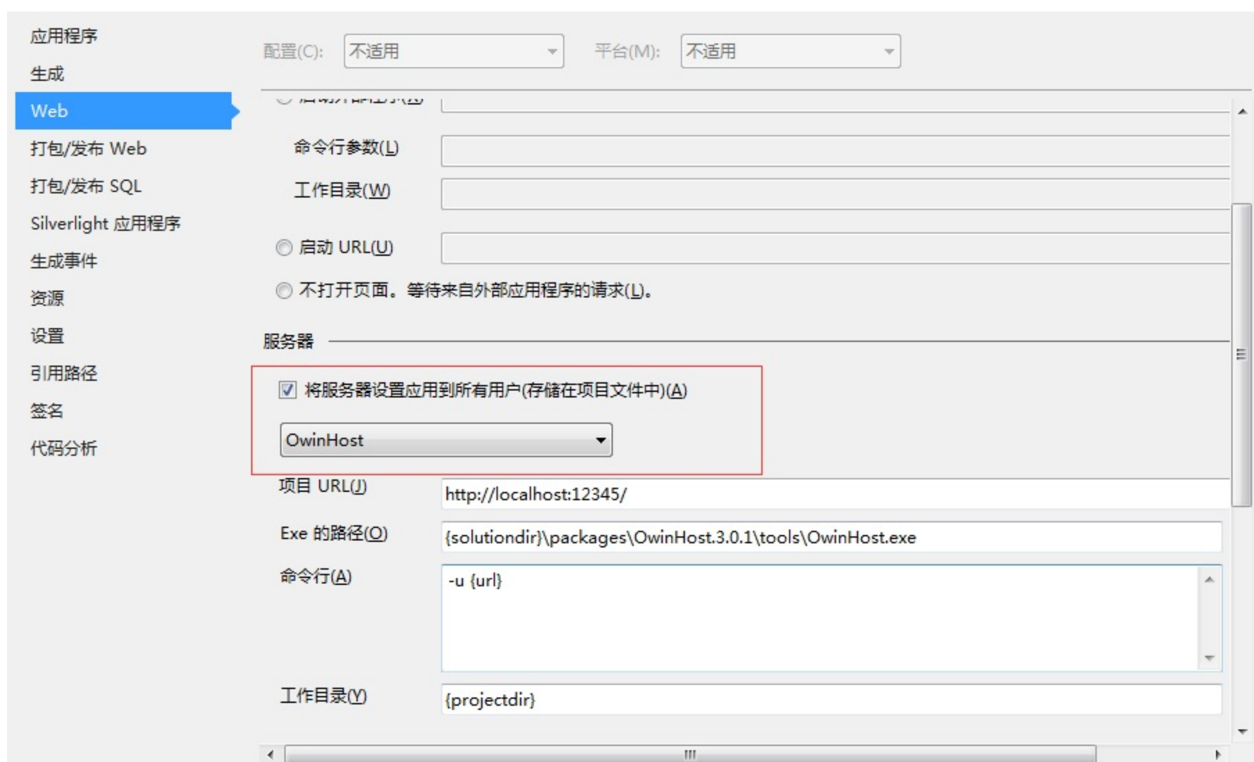
1. 创建一个空的Web应用程序

2. 通过Nuget安装OwinHost包

3. 添加OWIN Startup类，并添加如下代码：

```
 public void Configuration(IAppBuilder app)
        {
            // 有关如何配置应用程序的详细信息，请访问 http://go.microsoft
            app.Run(context =>
            {
               **context.Response.ContentType** **= "text/plain";
                return context.Response.WriteAsync("Hello, This is
            });
        }
```

4. 设置Web应用程序属性，将宿主从IIS Express更改为OwinHost。具体设置如下图所示：



然后运行该网站，你将在浏览器中看到“Hello, This is host in OwinHost.exe.”的字样。

五、让Asp.net网站运行在定义的Web服务器上

前面我们简单应用了Kanata支持的三种宿主方式。但如果我们想将我们的Asp.net网站运行到自定义的Web 服务器上该怎么办呢？朋友们，你们是否还记到，我在C#网络编程系列中，已经实现一个轻量的Web 服务器了。既然OWIN规范可以使得我们可以将Asp.net网站不再依赖于IIS Web 服务器，那自然我们就可以通过自定义Web 服务器，然后让Asp.net运行在我们自定义的Web 服务器上了。接下来让我们具体看看，如何实现Asp.net网站运行在我们自定义的Web 服务器上的。

1. 首先自定义Web 服务器。具体的实现代码如下所示：

```
using System.Net;
using System.Net.Sockets;
using AppFunc = Func<IDictionary<string, object>, Task>;

public class CustomServer
{
    public CustomServer()
    {
        // Create a configurable instance
    }

    public void Start(AppFunc next, IList<IDictionary<string, o
    {

        // 获得本机的Ip地址，即127.0.0.1
```

```
            IPAddress localaddress = IPAddress.Loopback;

            // 创建可以访问的断点，49155表示端口号，如果这里设置为0，表示使/
            IPEndPoint endpoint = new IPEndPoint(localaddress, 8888

            // 创建Tcp 监听器
            TcpListener tcpListener = new TcpListener(endpoint);

            // 启动监听
            tcpListener.Start();
            Console.WriteLine("Wait an connect Request...");
            while (true)
            {
                // 等待客户连接
                TcpClient client = tcpListener.AcceptTcpClient();
                if (client.Connected == true)
                {
                    // 输出已经建立连接
                    Console.WriteLine("Created connection");
                }

                // 获得一个网络流对象
                // 该网络流对象封装了Socket的输入和输出操作
                // 此时通过对网络流对象进行写入来返回响应消息
                // 通过对网络流对象进行读取来获得请求消息
                NetworkStream netstream = client.GetStream();
                // 把客户端的请求数据读入保存到一个数组中
                byte[] buffer = new byte[2048];

                int receivelength = netstream.Read(buffer, 0, 2048)
                string requeststring = Encoding.UTF8.GetString(buf

                // 在服务器端输出请求的消息
                Console.WriteLine(requeststring);

                // 服务器端做出相应内容
                // 响应的状态行
                string statusLine = "HTTP/1.1 200 OK\r\n";
                byte[] responseStatusLineBytes = Encoding.UTF8.GetE
                string responseBody = "<html><head><title>Default F
                string responseHeader =
                    string.Format(
                        "Content-Type: text/html; charset=UTf-8\r\n

                byte[] responseHeaderBytes = Encoding.UTF8.GetBytes
                byte[] responseBodyBytes = Encoding.UTF8.GetBytes(r

                // 写入状态行信息
                netstream.Write(responseStatusLineBytes, 0, respons
                // 写入回应的头部
                netstream.Write(responseHeaderBytes, 0, responseHea
                // 写入回应头部和内容之间的空行
                netstream.Write(new byte[] { 13, 10 }, 0, 2);
```

```
                // 写入回应的内容
                netstream.Write(responseBodyBytes, 0, responseBodyE

                // 关闭与客户端的连接
                client.Close();
                Console.ReadKey();
                break;
            }

            // 关闭服务器
            tcpListener.Stop();
        }
    }

using AppFunc = Func<IDictionary<string, object>, Task>;

    public static class OwinServerFactory
    {
        /// <summary>
        /// Optional. This gives the server the chance to tell the
        /// </summary>
        /// <param name="properties"></param>
        public static void Initialize(IDictionary<string, object> p
        {
            // TODO: Add Owin.Types.BuilderProperties for setting c

            // Consider adding a configurable object to the propert
            properties[typeof(CustomServer).FullName] = new CustomS
        }

        public static CustomServer Create(AppFunc app, IDictionary<
        {
            object obj;

            // Get the user configured server instance, if any.
            CustomServer server = null;
            if (properties.TryGetValue(typeof(CustomServer).FullNar
            {
                server = obj as CustomServer;
            }
            server = server ?? new CustomServer();

            // Get the address collection
            IList<IDictionary<string, object>> addresses = null;
            if (properties.TryGetValue("host.Addresses", out obj))
            {
                addresses = obj as IList<IDictionary<string, object
            }

            server.Start(app, addresses);

            return server;
```

```
        }
    }
```

2. 创建一个控制台应用程序来对Web应用程序进行自我宿主。

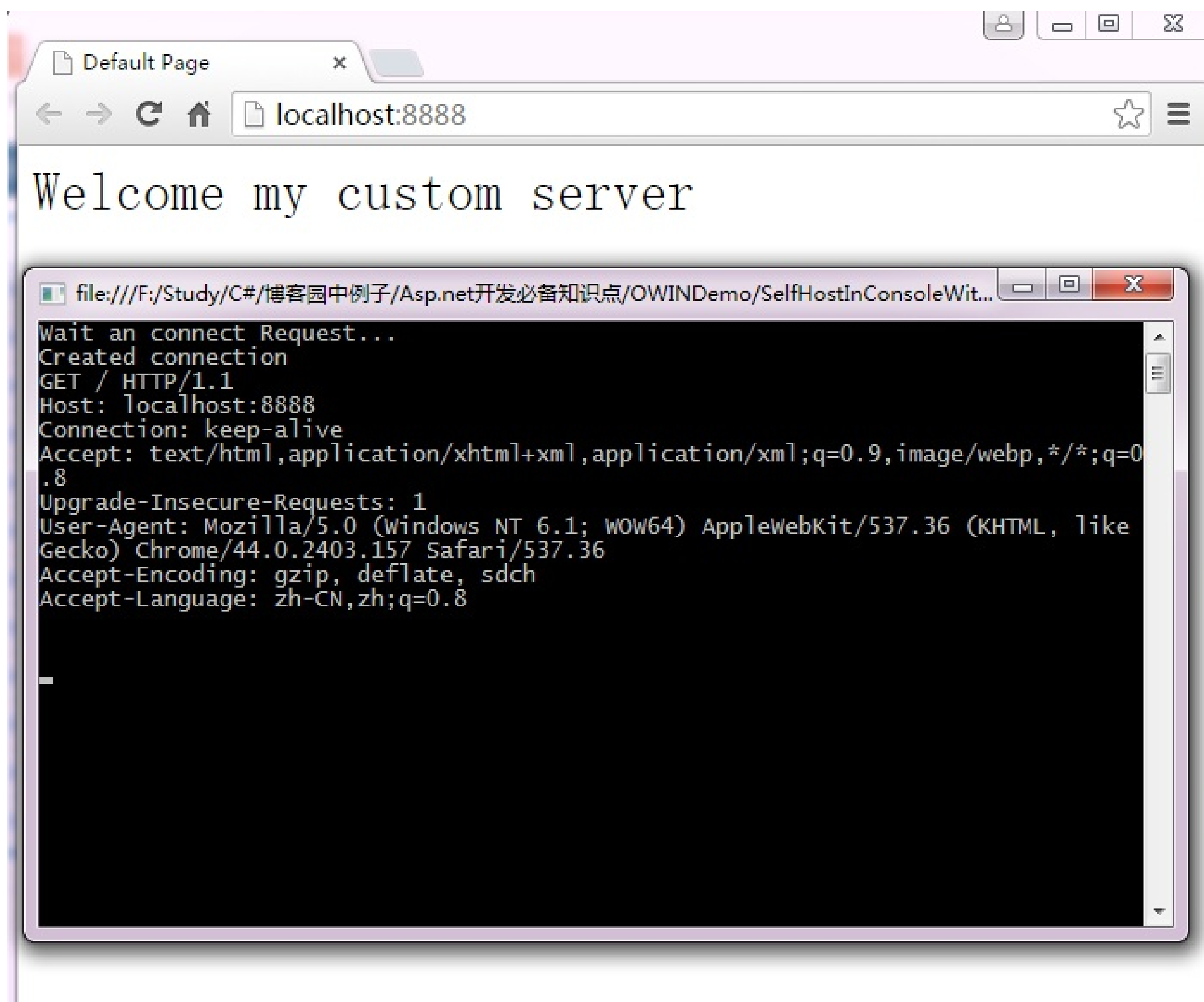
3. 通过Nuget添加“Microsoft.Owin.Hosting”包

4. 添加OWIN Startup类

5. 往Main方法中添加下面代码：

```
static void Main(string[] args)
        {
            using (WebApp.Start<Startup>(
            new StartOptions(url: "http://localhost:8888") { Server
            {
                Console.WriteLine("Started, Press any key to stop.'
                Console.ReadLine();
                Console.WriteLine("Stopped");
            }
        }
```

此时，运行该控制台程序，然后在浏览器中输入“localhost:8888”，你将看到如下结果：



到此，我们已经将Asp.net站点运行在我们自定义的Web 服务器上了。

六、总结

到这里，关于OWIN的介绍就到此结束了，接下来将介绍Asp.net最新的用户权限管理：Asp.net Identity的相关内容。

本文的所有源码下载：OWINDemo

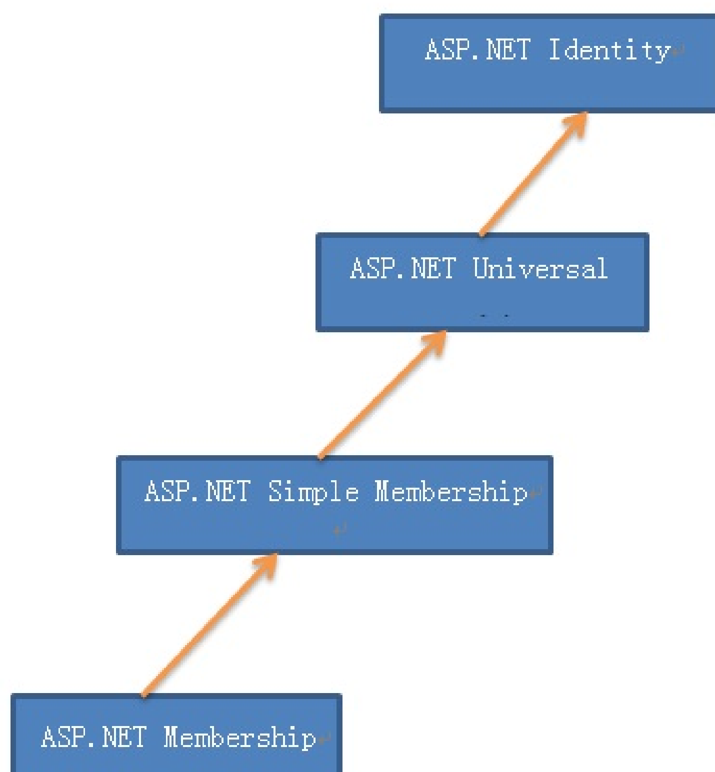
ASP.NET 开发必备知识点(2)：那些年追过的 ASP.NET权限管理

一、前言

在前一篇文章已经为大家介绍了OWIN和Katana，有了对他们的了解之后，才能更好地去学习Asp.net Identity，因为Asp.net Identity的实现集成了Owin。其实在Asp.net 2.0的时候，微软已经对用户权限管理进行了实现，其实现为Membership。由于之前的实现有很多限制，所以微软在Asp.net 4.5推出了Asp.net Identity。接下来，本篇文章将详细介绍下Asp.net Identity的实现。

二、Asp.net中用户权限管理发展历程

在前面我们已经说过，在Asp.net 2.0的时候，Asp.net中就已经实现了用户权限管理，所以，Asp.net 用户权限管理有其发展历程。下图就是Asp.net中权限管理的发展历程：



ASP.NET Membership

Asp.net Membership是在2005年的Asp.net 2.0引入的。Membership机制引入了表单验证（Form Authentication），以及一个用于存储用户名、密码和其他用户信息的SQL Server数据库。但它同样存在一些限制：

- 数据库只能使用SQL Server，难以对SQL Server Compact、SQL Azure、NoSQL支持。并且你想为用户表添加额外字段的话，此时你只能创建一个User的附加表。对于开发者来说，不能很好地自定义用户信息。
- 由于Asp.net Membership是基于表单进行验证的，因此无法支持OWIN。

ASP.NET Simple Membership

Asp.net Simple Membership是对Asp.net Membership的一次改进，它使得你可以更容易自定义用户信息。尽管如此，由于它依然是基于Asp.net Membership之上的，所以它仍然存在以下几点限制：

- 对非关系数据库支持不好。
- 不支持OWIN
- 对于已存在的Asp.net Membership Provider支持的不是很好，不利于扩展。

ASP.NET Universal Providers

Asp.net Universal Providers解决了前两者的一些问题，例如他支持存储用户在Azure SQL和SQL Server Compact数据库中。并且它基于EF code First实现的，所以它支持EF支持的所有数据库。但由于它依然是基于Asp.net Membreship基础架构实现的，所以仍然有些问题不能很好解决。所以它只解决了前两者的部分问题，其本身还存在一些限制：

- 对非关系数据库支持不好
- 不支持OWIN

三、Asp.net Identity 详细介绍

随着互联网的快速发展，从而非关系数据库也层出不穷，但之前的三者权限管理都对非关系数据库支持的不是很好，所以微软必须要实现一种新的权限管理机制，所以在。NET Framework中推出了Asp.net Identity。该套机制解决了之前的所有问题。Asp.net 具有如下特点：

- 可用于ASP.NET所有框架上，包括Asp.net MVC、Asp.net Web Forms、Web Pages、Asp.net Web API和SignalR。
- 可用于各种应用程序，包括Web应用、移动应用，Windows Store应用和混合架构应用。
- 用户信息的自定义
- 存储易于扩展：默认使用EF Code First存储在SQL Server数据库中，但可以很好地扩展到SharePoint、Azure SQL和NoSQL 数据库中。
- 支持单元测试
- 提供了Role Provider，使创建和管理变得简单
- 支持面向Claims的身份验证（即：支持基于声明的身份验证），前面的三者都是基于表单的身份验证。
- 支持社交账号的登录，支持Facebook,Microsoft账户、Twitter，Google、QQ等社交账户。
- 支持Windows Azure Active Directory账号登录功能

- 支持OWIN。
- 通过Nuget发布，这样能让Asp.net 团队更好地修复Bug和迭代新功能，并在第一个时间进行发布。将其与System.Web.dll程序集解耦。

四、Asp.net Identity内部实现机制

从上面对Asp.net Identity的介绍可以发现，它确实解决了之前的所有问题，那它是如何做到的呢？要知道其实现机制并不难，因为Asp.net Identity已经开源，我们可以到其站点下载其源码研究即可，其开源地址为：http://aspnetidentity.codeplex.com/。这里简单的分析它注册和登录的功能的内部实现。

首先使用VS2013创建一个Asp.net MVC站点，此时网站的用户授权和认证模块的代码的实现VS已经帮我们添加好了，我们只需要找到对应的注册和登录功能对其进行分析，从而明白Asp.net Identity是如何帮完成这两个功能的。

首先，我们找到Accout控制器中注册功能的实现代码：

```
public async Task<ActionResult> Register(RegisterViewModel model)
        {
            if (ModelState.IsValid)
            {
                var user = new ApplicationUser { UserName = model.E
                var result = await UserManager.CreateAsync(user, mo
                if (result.Succeeded)
                {
                    await SignInManager.SignInAsync(user, isPersist

                    // 有关如何启用帐户确认和密码重置的详细信息，请访问 htt
                    // 发送包含此链接的电子邮件
                    // string code = await UserManager.GenerateEmai
                    // var callbackUrl = Url.Action("ConfirmEmail",
                    // await UserManager.SendEmailAsync(user.Id, "确

                    return RedirectToAction("Index", "Home");
                }
                AddErrors(result);
            }

            // 如果我们进行到这一步时某个地方出错，则重新显示表单
            return View(model);
        }
```

从上面代码的方法名可以看出，完成用户注册的主要实现在于UserManager.CreateAsync方法上，这个方法实现真是在Asp.net Identity帮我们实现，接下来到我们下载的源码来查看该方法的实现。具体的源码实现如下所示:

```
public virtual async Task<IdentityResult> CreateAsync(TUser user, s
        {
            ThrowIfDisposed();
            var passwordStore = GetPasswordStore();
            if (user == null)
            {
                throw new ArgumentNullException("user");
            }
            if (password == null)
            {
                throw new ArgumentNullException("password");
            }
            // UpdatePassword对密码进行Hash加密
            var result = await UpdatePassword(passwordStore, user, 
            if (!result.Succeeded)
            {
                return result;
            }
            // 注册功能的实现
            return await CreateAsync(user).WithCurrentCulture();
        }

     public virtual async Task<IdentityResult> CreateAsync(TUser us
        {
            ThrowIfDisposed();
            await UpdateSecurityStampInternal(user).WithCurrentCult
            var result = await UserValidator.ValidateAsync(user).Wi
            if (!result.Succeeded)
            {
                return result;
            }
            if (UserLockoutEnabledByDefault && SupportsUserLockout)
            {
                await GetUserLockoutStore().SetLockoutEnabledAsync(
            }

            // 调用IUserStore的CreateAsync完成用户注册
            await Store.CreateAsync(user).WithCurrentCulture();
            return IdentityResult.Success;
        }

         // UserStore中CreateAsync的实现
        public virtual async Task CreateAsync(TUser user)
        {
            ThrowIfDisposed();
            if (user == null)
            {
                throw new ArgumentNullException("user");
            }

            // 将实体添加进DbSet<User>集合中
            _userStore.Create(user);
```

```
            // 调用SaveChanges将用户保存到数据库中
            await SaveChanges().WithCurrentCulture();
        }
```

看到这里是不是豁然开朗了很多的，其实Asp.net Identity内部注册功能的实现，我们完全可以自己来实现。其实现简单的说就是：

1. 对用户提交的数据进行验证

2. 对密码进行加密保存

3. 调用Microsoft.AspNet.Identity.EntityFramework命名空间下的UserStore类的CreateAsync方法将用户进行持久化。

4. UserStore类中的CreateAsync方法的实现也就是DbSet<User>.Add(entity)和SaveChanges()方法将对象持久化。

到这里还有一个问题，其实上面代码调用的是IUserStore接口中的CreateAsync方法，但具体的IUserStore对象是怎么注入进去的呢？

你带着这个疑惑去Asp.net MVC站点中去寻找其注入代码。此时，你可以发现在Startup.Auth.cs和Start.cs文件中有如下实现：

```
    // Startup.cs 文件
    public partial class Startup
    {
        public void Configuration(IAppBuilder app)
        {
            ConfigureAuth(app);
        }
    }

    // Start.Auth.cs文件
     public void ConfigureAuth(IAppBuilder app)
        {
            // 配置数据库上下文、用户管理器和登录管理器，以便为每个请求使用单
            // .........
            app.CreatePerOwinContext<ApplicationUserManager>(Applic
           // ........
    }

    // IdentityConfig.cs文件
      public static ApplicationUserManager Create(IdentityFactoryOp
        {
            var manager = new **ApplicationUserManager(new UserStor
    }
```

看到上面标注红色的代码了吗，这里就是将UserStore注入的地方。看到这里，你是不是没有任何疑惑了。对于登录功能的实现大家同样可以按照这样的方法去探索。本来想一起分析下的，后面想想，还是留给大家去探索吧。

五、从Asp.net Identity内部实现学会项目分层架构

其实，在我们平时工作，只要学会如何使用Asp.net Identity机制来完成对应功能。那我们为什么还要研究其源码实现呢？我觉得有两点：

1. 研究源码实现，可以让你对其实现原理有一个深刻的理解，对于分析出现的问题有极大的好处。因为只有你了解其实现原理，写功能模块才能更加自信，处理出现的问题才会比别人快。
2. 除了第一点之外，研究源码还有一个重要的作用就是学习源码作者的项目分层和代码分离。在现实生活中，有很多朋友抱怨出现瓶颈了，无法提高，因为平常工作中一般都是去写堆功能的代码，觉得对能力没什么提高。此时你完全可以去研究微软开源的代码，通过研究源码来学习大牛们是如何将项目做到低耦合高内聚的，学习大牛们是如何做到代码分离的。然后再讲学习到的内容应用于工作，相信这样的一个过程下来，你不想提高都不行了。渐渐地你会觉得自己也可以完成一个开源框架。

上面介绍了研究源码的两大作用，那我们从Asp.net Identity内部实现中又学到了什么呢？

通过第四部分的代码分析，Asp.net Identity中注册功能的实现主要分为的4点中，我们可以学到如下几点：

1. 关注点的分离。Asp.net Identity注册功能中，将用户输入以及密码加密等代码实现都分离到具体的类中进行实现，而不是将其放在UserManager这个类中。这充分体现关注点分离原则
2. 针对接口编程原则。Asp.net Identity内部实现中，都是针对于接口编程，每个类中依赖都是接口，并没有依赖与具体类。从而降低代码之间的耦合。
3. 实现了依赖注入。Asp.net Identity具体实现是通过在调用端通过依赖注入的方式进行注入。
4. 项目分层架构。Asp.net Identity注册功能。AccountController首先调用UserManager的CreateAsync，而UserManager的CreateAsync又调用了IUserStore中的CreateAsync方法来通过调用EF的DbContext来完成数据的持久化。从这个调用过程和类之间的关系可以看出，这真是领域驱动设计的分层体现。领域驱动设计中设计4层，分别是UI层、应用层、领域层和基础设施层。其中UI层对应的就是AccountController类，应用层对应的就是UserManager类、领域层就是具体的User实体、基础设施层对应的就是IUserStore（准确地说，基础设施层中的仓储对应着IUserStore）。上面对应的项目分层，其实每个层中代码的实现都可以按照这个模式去实现。这点在ABP Web框架中得到了很好的实现：https://github.com/aspnetboilerplate/aspnetboilerplate。

所以，如果你觉得你现在的工作得到提高的话，完全不需要去什么群里咨询其他的推荐什么书籍什么，从现在开始就开始研究源码吧。如果不知道研究什么源码的话，完全可以从微软的一些开源代码开始，例如就从Asp.net Identity源码开始，不

要担心研究完之后，还是怕不能提高，只要你理解和领悟了，你不想提高都难。另外再推荐大家研究下ABP的实现，我最近就在研究它，希望理解透彻之后，再写一个小的Web框架来巩固自己的研究。到时候也会把自己的一些研究心得分享到这里。

六、总结

到这里，本篇文章的介绍就结束了，希望这篇文章可以帮助朋友对微软的用户权限管理框架有进一步的了解，以及希望哪些想提高的朋友，从现在开始就来和我一起来研究微软的开源框架和ABP框架吧。记得研究之后，分享到这里与大家共享哦。

ASP.NET中实现回调

一、引言

在ASp.NET网页的默认模型中，用户通过单击按钮或其他操作的方式来提交页面，此时客户端将当前页面表单中的所有数据（包括一些自动生成的隐藏域）都提交到服务器端，服务器将重新实例化一个当前页面类的实例来响应这个请求，然后将整个页面的内容重新发送到客户端。这种处理方式对运行结果没什么影响，但页回发会导致处理开销，从而降低性能，且会让用户不得不等待处理并重新创建页，有时候，我们仅仅只需要传递部分数据而不需要提交整个表单，这种默认的处理方式（指的是提交整个表单进行回发方式）显得有点小题大做了，解决办法主要有三种：纯JS实现、Ajax技术和回调技术，在这里仅仅介绍下Asp.net回调技术的实现。（回调的本质其实就是**Ajax**调用，之所以这么说是因为我们使用Asp.net中的类来实现回调，Asp.net中类会帮我们做Ajax的操作）。

二、实现步骤

使用回调技术来实现无刷新页面的要点是：

1. 让当前页面实现**ICallbackEventHandler**.aspx)接口，该接口定义了两个方法：GetCallbackResult.aspx) 方法和RaiseCallbackEvent.aspx)方法，其中，GetCallbackResult方法的作用是返回以控件为目标的回调方法的结果；RaiseCallbackEvent方法是处理以控件为目标的回调方法.
2. 为当前页面提供2个JS脚本，一个是客户端调用服务器端方法成功后要执行的客户端方法，一个是客户端调用服务器端方法失败后要执行的客户端方法。

具体测试页面代码为：

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Register

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "htt

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>用户注册</title>
    <script language="javascript">
       // 调用服务器端成功时调用的客户端方法
        function Success(arg, context) {
            document.getElementById("message").innerHTML = arg;
        }
        // 调用服务器端失败时调用的客户端方法
        function Error(arg, context) {
            document.getElementById("message").innerHTML = "发生异常
        }
    </script>
</head>
<body>
    <form id="form1" runat="server">
    <div>
    <div>
    用户名：
    <input type="text" id="txtUserName" onblur="CallServerMethod(tx
    <span id="message" style="color:Red"></span>
    </div>
    <div>
    密码：
    <input type="text" id="txtpassword" style="margin-left:15px" />
    </div>
    </div>
    </form>
</body>
</html>
```

后台CS代码为：

```
using System;
using System.Web.UI;

namespace ASPNETClientCallBackWithoutPostBack
{
    public partial class Register : System.Web.UI.Page, ICallbackE
    {
        string result=string.Empty;

        protected void Page_Load(object sender, EventArgs e)
        {
            // 获得当前页的ClientScriptManager对象，该对象用于管理客户端
            ClientScriptManager clientScriptManager = Page.ClientSc

            // 获取回调引用
            // 执行下面代码会在客户端生成WebForm_DoCallback方法，调用他来
            string reference = clientScriptManager.GetCallbackEvent
            string callBackScript = "function CallServerMethod(arg,

            // 向当前页面注册客户端脚本
            // CallServerMethod是要注册的客户端脚本的键
            clientScriptManager.RegisterClientScriptBlock(this.Get
        }

        /// <summary>
        /// 服务器端运行的回调方法
        /// </summary>
        /// <param name="eventArgument"></param>
        public void RaiseCallbackEvent(string eventArgument)
        {
            if (eventArgument.ToLower().IndexOf("admin") != -1)
            {
                result = eventArgument + "用户已注册";
            }
            else
            {
                result = eventArgument + "可以注册";
            }
        }

        /// <summary>
        /// 返回回调方法的执行结果
        /// </summary>
        public string GetCallbackResult()
        {
            return result;
        }
    }
}
```

当我们在浏览器中查看上面Asp.net页面时，Asp.net页面会经过服务器端Page类的处理生成标准的HTML代码，具体代码如下：

```
<html xmlns="http://www.w3.org/1999/xhtml"><head><title>
    用户注册
</title>
    <script language="javascript">
       // 调用服务器端成功时调用的客户端方法
        function Success(arg, context) {
            document.getElementById("message").innerHTML = arg;
        }
        // 调用服务器端失败时调用的客户端方法
        function Error(arg, context) {
            document.getElementById("message").innerHTML = "发生异常
        }
    </script>
</head>
<body>
    <form method="post" action="Register.aspx" id="form1">
<div class="aspNetHidden">
<input type="hidden" name="__EVENTTARGET" id="__EVENTTARGET" value=
<input type="hidden" name="__EVENTARGUMENT" id="__EVENTARGUMENT" va
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE" value="/wE
</div>

**// 在生成的HTML代码中多了几段JS代码块**
**// 这部分代码是每个Asp.net页面发送到客户端都会生成的，用于提交当前表单**
**// eventTarget表示激发提交时间的控件，eventArgument表示发生该事件时的参数
<script type="text/javascript">
//<![CDATA[
var theForm = document.forms['form1'];
if (!theForm) {
    theForm = document.form1;
}
function __doPostBack(eventTarget, eventArgument) {
    if (!theForm.onsubmit || (theForm.onsubmit() != false)) {
        theForm.__EVENTTARGET.value = eventTarget;
        theForm.__EVENTARGUMENT.value = eventArgument;
        theForm.submit();
    }
}
//]]>
</script>

**// 这部分代码用来生成用于Ajax调用的JS脚本，其源码中有WebForm_DoCallback7
<script src="/WebResource.axd?d=Okp1JZTDECHos0RqI93uiUGmigRVKnpI1Gr

**// 这部分代码是由服务端代码生成的，因为我们在后台代码中使用ClientScriptMar
<script type="text/javascript">
//<![CDATA[
function CallServerMethod(arg, context){WebForm_DoCallback('__Page
</script>
```

```
    <div>
    <div>
    用户名：
    <input type="text" id="txtUserName" onblur="CallServerMethod(tx
    <span id="message" style="color:Red"></span>
    </div>
    <div>
    密码：
    <input type="text" id="txtpassword" style="margin-left:15px">
    </div>
    </div>

**// WebForm—InitCallback方法的定义也在幕后生成的脚本文件中，脚本代码可以
<script type="text/javascript">
//<![CDATA[

WebForm_InitCallback();//]]>
</script>
</form>

</body></html>
```

三、运行结果

下面就看看上面代码实现的无刷新回调的效果：

四、小结

因为最近一段时间在学习Asp.net的内容，这里记录下一些学习过程中个人觉得比较重要的内容，希望对其他一些朋友有所帮助，关于更多关于客户端脚本进行Asp.NET 网页编程的内容参考MSDN：

http://msdn.microsoft.com/zh-cn/library/50b7y38h(v=vs.80).aspx.aspx)。在本文章中Asp.NET的回调技术的实现其本质就是是微软帮我们实现的一种Ajax实现罢了（具体原因可以查看WebForm_DoCallback源码就明白了）。

源码地址：http://files.cnblogs.com/zhili/ASPNETClientCallBackWithoutPostBack.rar

跟我一起学WCF

跟我一起学WCF(1)——MSMQ消息队列

一、引言

Windows Communication Foundation(WCF)是Microsoft为构建面向服务的应用程序而提供的统一编程模型，该服务模型提供了支持松散耦合和版本管理的序列化功能，并提供了与消息队列（MSMQ）、COM+、Asp.net Web服务、.NET Remoting等微软现有的分布式系统技术。利用WCF平台，开发人员可以很方便地构建面向服务的应用程序（SOA）。可以认为，WCF是对之前现有的分布式技术（指的是MSMQ、.NET Remoting和Web 服务等技术）的集成和扩展，既然这样，我们就有必要首先了解下之前分布式技术，只有这样才能更深刻地明白WCF所带来的好处。今天就分享下MSMQ这种分布式技术。

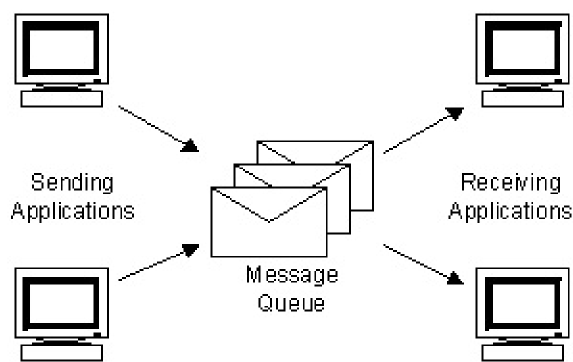
二、MSMQ的介绍

MSMQ全称是Microsoft Message Queue——微软消息队列。它是一种异步传输模式，可以在不同的应用之间实现相互通信，相互通信的应用可以分布在同一台机器上，也可以分布于相连的网络空间中的任一位置。

2.1 MSMQ 工作原理

MSMQ的实现原理是：消息的发送者把自己想要发送的信息放入一个容器，然后把它保存到一个系统公用空间的消息队列中，本地或异地的消息接收程序再从该队列中取出发给它的消息进行处理。

消息队列是一个公用存储空间，它可以存在于内存中或物理文件中，因此，消息以两种方式发送，即快递方式和可恢复模式。它们的区别是消息存储位置的不同，快递方式，为了消息的快速传递，所以把消息放置在内存中，而不放在物理磁盘上，以获得较高的处理能力；而可恢复模式在传送过程的每一步骤中，都把消息写入物理磁盘上，这样当保存消息队列的机器发生故障而重新启动后，可以把发送的消息恢复到故障发送之前的状态，以获得更好的消息恢复能力。消息队列可以放在发送方、接收方所在的机器上，也可以单独放置在另外一台机器上。另外，采用消息队列机制，发送方不必要担心接收方是否启动，是否发生故障等因素，只要消息成功发送出去，就可以认为处理完成，而实际上对方可能甚至未开机，或者实际消息传递到对方可能在第二天。MSMQ机制类似QQ消息传递机制。下图演示了MSMQ的实现原理。



MSMQ中主要有两个概念。

- 一个是消息Message：Message是通信双方需要传递的消息，它可以是文本、图片、视频等。消息包含发送和接收者的标识，只有指定的用户才能取得消息。
- 一个是队列Queue：用来保存消息的存储空间，MSMQ中主要包括以下几种队列类型：
    - 公共队列：在整个消息队列网络中复制，有可能由网络连接的所有站点访问。路径格式为：机器名称\队列名称
    - 专用队列（或叫私有队列）：不在整个网络中发布，它们仅在所驻留的本地计算机上可用，专用队列只能由知道队列的完整路径名称或标签的应用程序访问。路径格式为：机器名称\Private$\队列名称
    - 日志队列：包含确认在给定"消息队列中发送的消息回执消息"。路径格式为：机器名称\队列名称\Journal$
    - 机器日志队列对应的格式为：机器名称\Journal$；
    - 机器死信队列对应的格式为：机器名称\Deadletter$；
    - 机器信道死信队列对应的格式为：机器名称\XactDeadletter$。

2.2 队列引用说明

当创建了一个MessageQueue实例之后，就应指明和哪个队列进行通信，在.NET中有3种访问指定消息队列的方法：

- 使用路径，消息队列的路径被机器名和队列名唯一确定，所以可以用消息队列路径来指明使用的消息队列。
- 使用格式名（format name），它是由MSMQ在消息队列创建时生成的唯一标识，个使命不由用户指定，而是由队列管理者自动生成的GUID。
- 使用标识名（label），它是消息队列创建时由消息管理者指定的带有意义的名字。

三、消息队列的优缺点

采用消息队列的好处是：由于是异步通信，无论是发送方还是接收方都不同等待对方返回成功消息，就可以执行余下的代码，大大提高了处理的能力；在信息传递过程中，具有故障恢复能力；MSMQ的消息传递机制使得通信的双方具有不同的物理

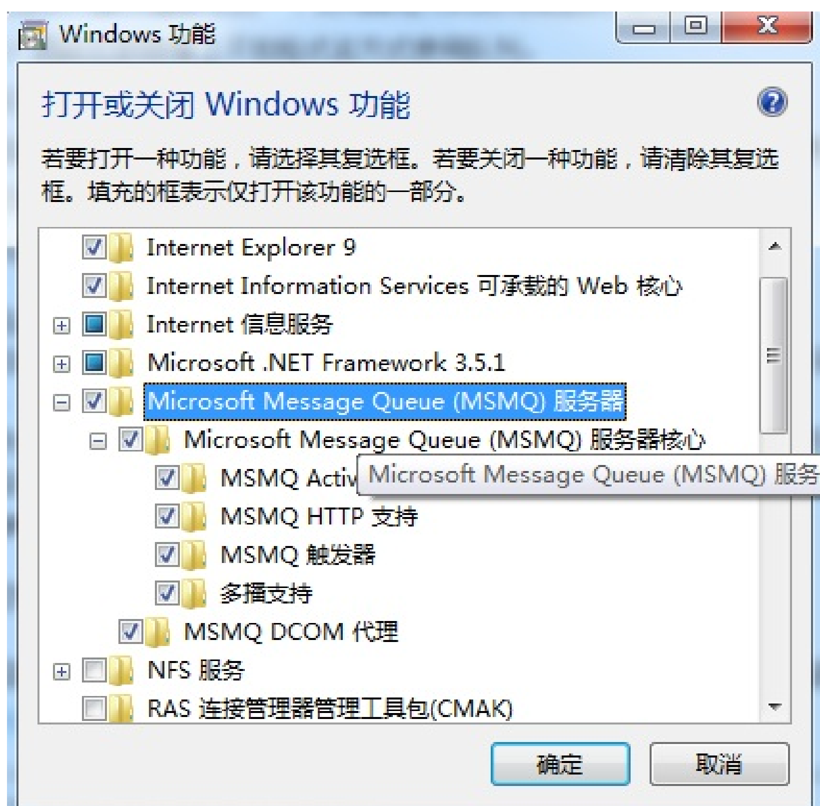
平台成为可能。

消息队列缺点是：不适合Client需要Server端实时交互情况，大量请求时候，响应可能延迟。对于客户端，必须是Windows系统。可以通过连接器跟其他的非微软技术集成。

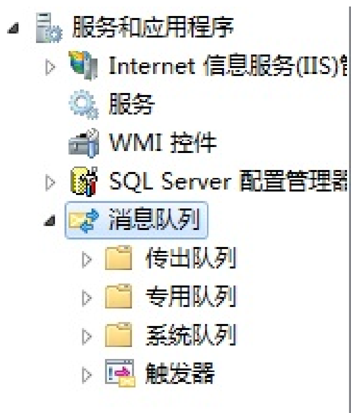
四、利用MSMQ开发分布式应用

4.1 环境准备

要想在.NET平台进行MSMQ的开发，需要安装消息队列，你需要打开控制面板->程序->打开或关闭Windows功能，勾选消息队列服务所有选项，具体操作如下图所示：



勾选完之后点击确定之后，可以在我的电脑->管理->服务和应用程序->消息队列 看到下面的图：



看到上面这个图代表你已经成功配置了MSMQ的开发环境，下面就可以使用Visual Studio 进行开发。注意，对特定类型队列的操作代码，一定要成功安装对应的队列类型。

4.2 使用MSMQ开发分布式应用

首先，实现服务器端。创建一个控制台项目，添加System.Messaging引用，因为消息队列的类全部封装在System.Messaging.dll程序集里。具体服务端的代码如下：

```
 1 using System;
 2 using System.Messaging;
 3 
 4 namespace MSMQServer
 5 {
 6     class Program
 7     {
 8         static void Main(string[] args)
 9         {
10             // 创建一个公共队列,公共队列只能创建在域环境里
11             //if (!MessageQueue.Exists(@".\LearningHardMSMQ")) /
12             //{
13             //    using (MessageQueue mq = MessageQueue.Create(@
14             //    {
15             //        mq.Label = "LearningHardQueue"; // 设置队列
16             //        Console.WriteLine("已经创建了一个公共队列");
17             //        Console.WriteLine("路径为:{0}", mq.Path);
18             //        Console.WriteLine("队列名字为:{0}", mq.Queu
19             //        mq.Send("MSMQ Message", "Leaning Hard"); /
20             //    }
21             //}
22 
23             //if (MessageQueue.Exists(@".\Private$\LearningHardM
24             //{
25                   // 删除消息队列
26             //    MessageQueue.Delete(@".\Private$\LearningHardM
27             //}
28             // 创建一个私有消息队列
```

```
29                 if (!MessageQueue.Exists(@".\Private$\LearningHardMS
30                 {
31                     using (MessageQueue mq = MessageQueue.Create(@".
32                     {
33                         mq.Label = "LearningHardPrivateQueue";
34                         Console.WriteLine("已经创建了一个私有队列");
35                         Console.WriteLine("路径为:{0}", mq.Path);
36                         Console.WriteLine("私有队列名字为:{0}", mq.Que
37                         mq.Send("MSMQ Private Message", "Leaning Har
38                     }
39                 }
40 
41                 // 遍历所有的公共消息队列
42                 //foreach (MessageQueue mq in MessageQueue.GetPublic
43                 //{
44                 //    mq.Send("Sending MSMQ public message" + DateTi
45                 //    Console.WriteLine("Public Message is sent to {
46                 //}
47 
48                 if (MessageQueue.Exists(@".\Private$\LearningHardMSM
49                 {
50                     // 获得私有消息队列
51                     MessageQueue mq = new MessageQueue(@".\Private$\
52                     mq.Send("Sending MSMQ private message" + DateTim
53                     Console.WriteLine("Private Message is sent to {0
54                 }
55 
56                 Console.Read();
57             }
58         }
59 }
```

服务器端代码需要注意的是，公共队列只能在域环境中创建，由于我的个人电脑没有加入域环境，所以不能创建公共队列，从开始的消息队列的截图也可以看出，在图中并没有安装公共队列。

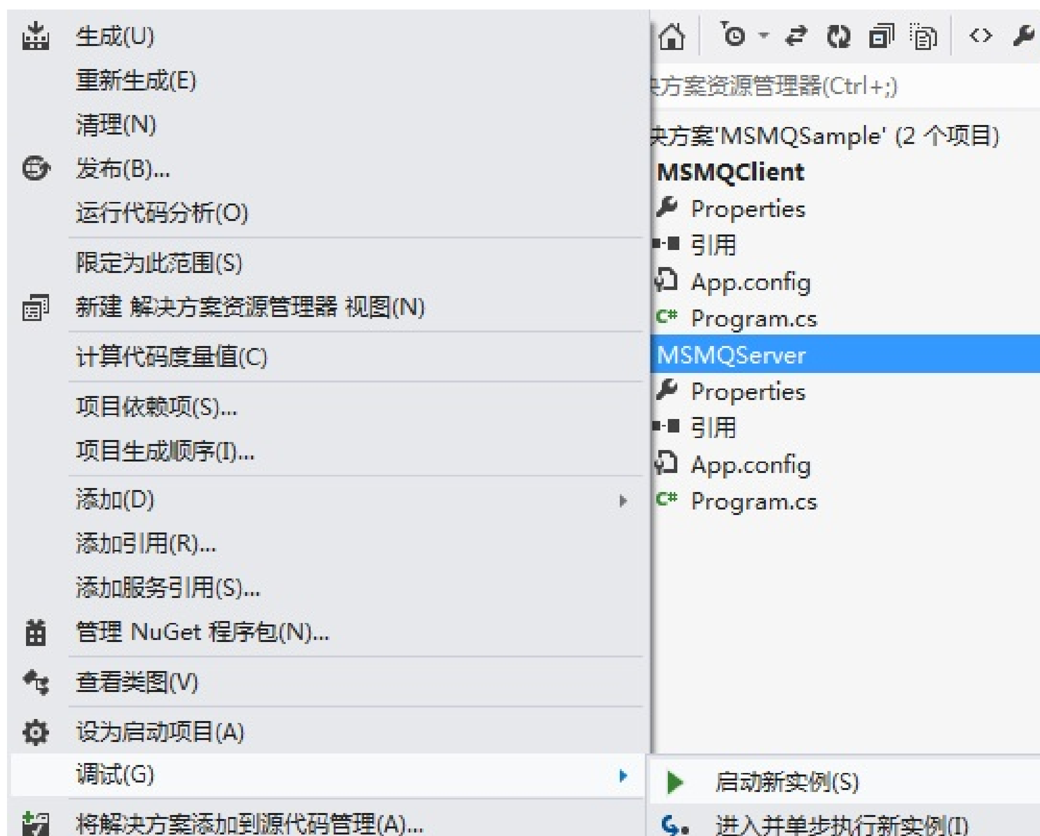
实现完服务器端之后，自然就是完成客户端。MSMQ程序的原理主要是：服务器端把消息发送到共享的消息队列中，然后，客户端从这个共享的消息队列中取出消息进行处理。具体客户端的实现代码如下所示：

```
1 using System;
2 using System.Messaging; // 需要添加System.Messaging引用
3
4 namespace MSMQClient
5 {
6     class Program
7     {
8         static void Main(string[] args)
9         {
10            if (MessageQueue.Exists(@".\Private$\LearningHardMSM
11            {
12                // 创建消息队列对象
13                using (MessageQueue mq = new MessageQueue(@".\Pr
14                {
15                    // 设置消息队列的格式化器
16                    mq.Formatter = new XmlMessageFormatter(new s
17                    foreach (Message msg in mq.GetAllMessages()
18                    {
19                        Console.WriteLine("Received Private Mess
20                    }
21
22                    Message firstmsg = mq.Receive(); // 获得消息队
23                    Console.WriteLine("Received The first Privat
24                }
25            }
26            Console.Read();
27        }
28    }
29 }
```

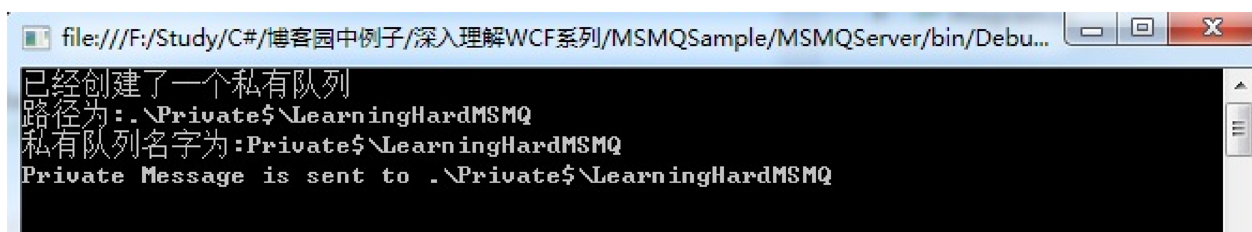
4.3 运行演示

经过上面步骤，我们已经完成了MSMQ分布式程序的实现了，下面看看如何运行该程序来查看效果。

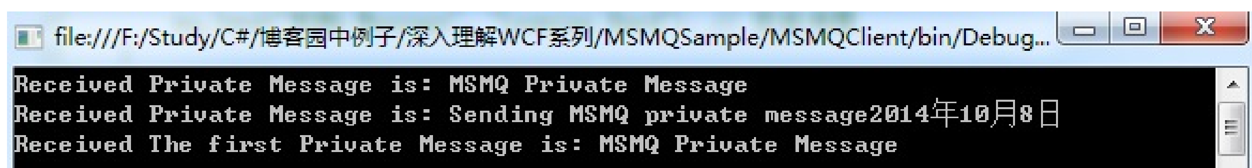
首先，自然要启动服务器，右键MSMQServer项目->调试->启动新实例来启动服务器，具体步骤如下图所示：



运行成功之后，你将到服务器发送消息成功的控制台界面，效果图如下所示：



接下来运行客户端来从消息队列中取得消息并显示在控制台中，采用和服务器相同的方式来启动客户端，右键MSMQClient->调试->启动新实例，看到客户端的效果如下图所示：



从上图可以看出，客户端确实成功地取得了消息队列中的消息。

以上MSMQ程序需要特别注意是：MessageQueue.Receive().aspx)是取出消息队列中队列中的第一条消息，并从消息队列中移除它（MSDN中文翻译上是错误，MSDN写的是不移除，而英文原文是移除），而实际结果也是移除的，如果你再运行一次客户端时，你会发现消息队列中只有一条消息，具体运行效果如下图所示：



五、总结

到这里，MSMQ的内容就分享结束，其MSMQ的实现原理也非常简单，一句话概括就是服务器把消息放在一个公共的地方，这个地方叫做消息队列，而其他客户端可以从这个地方取出消息进行处理。下一章将分享.NET 平台上另外一种分布式技术——.NET Remoting。

本文的示例代码文件：MSMQSample。

跟我一起学WCF(2)——利用.NET Remoting技术开发分布式应用

一、引言

上一篇博文分享了消息队列（MSMQ）技术来实现分布式应用，在这篇博文继续分享下.NET平台下另一种分布式技术——.NET Remoting。

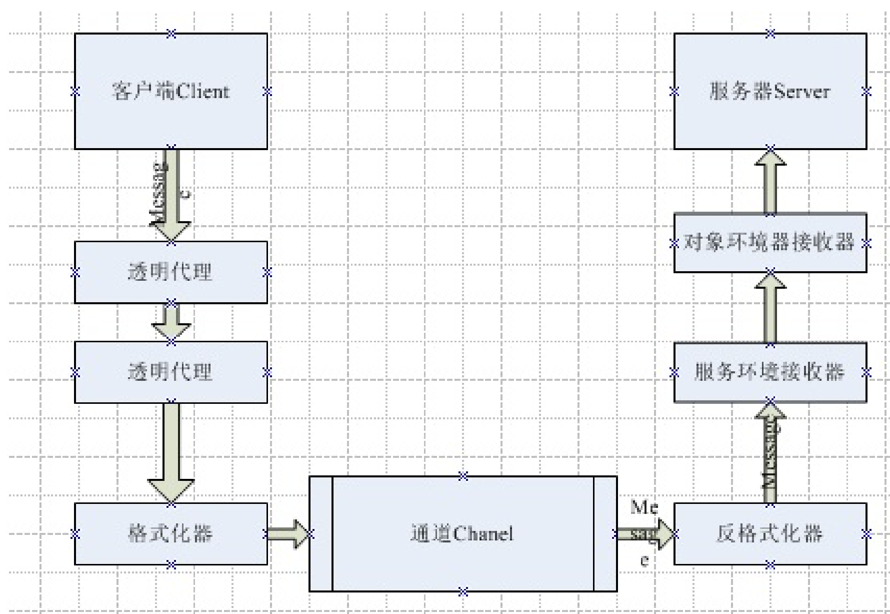
二、.NET Remoting 介绍

2.1 .NET Remoting简介

.NET REmoting与MSMQ不同，它不支持离线可得，另外只适合.NET平台的程序进行通信。它提供了一种允许对象通过应用程序域与另一个对象进行交互的框架。.NET 应用程序都在一个主应用程序域中执行的，在一个应用程序域中的代码不能访问另一个应用程序域的数据，然而在某些情况下，我们需要跨应用程序域，与另外的应用程序域进行通信，这时候就可以采用.NET Remoting技术来实现与另一个程序域中的对象进行交互。

2.2 .NET Remoting基本原理

.NET Remoting技术是通过通道来实现两个应用程序之间对象的通信的。首先，客户端通过Remoting技术，访问通道来获得服务器端对象，再通过代理解析为客户端对象，也称作透明代理，此时获得客户端对象只是服务器对象的一个引用。这既保证了客户端和服务端有关对象的松散耦合，同时优化了通信的性能。在这个过程中，当客户端通过透明代理来调用远程对象的方法时，此时会将调用封装到一个消息对象中，该消息对象包括远程对象信息，被调用的方法名和参数，然后透明代理会将调用委托给真实代理（RealProxy.aspx)对象）的Invoke方法来生成一个IMethodCallMessage.aspx)，接着通过序列化把这个消息对象序列化成数据流发送到通道，通道会把数据流传送到服务器端。当服务器接收到经过格式化的数据之后，首先从中通过反序列化来还原消息对象，之后在服务器端来激活远程对象，并调用对应的方法，而方法的返回结果过程则是按照之前的方法反向重复一遍，具体的实现原理图如下所示：



2.3 .NET Remoting几个重要概念

上面简单介绍了下.NET Remoting实现分布式应用程序的基本原理，这里介绍下在.NET Remoting中涉及的几个重要概念。

1. 远程对象：是运行在服务器端的对象，客户端不能直接调用，由于.NET Remoting传递的对象是以引用的方式，因此所传递的远程对象必须继承MarshalByRefObject类，这个类可以使远程对象在.NET Remoting应用通信中使用，支持对象的跨域边界访问。
2. 远程对象的激活方式：在访问服务器端的一个对象实例之前，必须通过一个名为Activation的进程创建它并进行初始化。这种客户端通过通道来创建远程对象的方式称为对象的激活。在.NET Remoting中，远程对象的激活分为两大类：服务器端激活和客户端激活。

3. 服务器端激活，又叫做WellKnow（知名对象）激活模式，为什么称为知名对象激活模式呢？是因为服务应用程序在激活对象实例之前会在一个众所周知的统一资源标示符（URI）上发布这个类型，然后该服务器进行会为此类型配置一个WellKnow对象，并根据指定的端口或地址来发布对象。.NET Remoting把服务器端激活又分为SingleTon模式和SingleCall模式两种。

SingleTon模式：此为有状态模式。如果设置为SingleTon激活模式，则.NET Remoting将为所有客户端建立同一个对象实例。当对象处于活动状态时，SingleTon实例会处理所有后来的客户端访问请求，而不管它们是同一个客户端，还是其他客户端。SingleTon实例将在方法调用中一直维护其状态，类似static成员的概念

SingleCall模式：是一种无状态模式。一旦设置为SingleCall模式，则当客户端调用远程对象的方法时，Remoting会为每一个客户端建立一个远程对象实例，对象实例的销毁则是由GC自动管理。类似实例成员的概念。

- 客户端激活：与Wellknow模式不同，。NET Remoting在激活每个对象实例的时候，会给每个客户端激活的类型指派一个URI。客户端激活模式一旦获得客

户端的请求，将为每一个客户端都建立一个实例引用。SingleCall模式与客户端激活模式的区别有：首先，对象实例创建的时间不同。客户端激活方式是客户一旦发出调用请求就实例化，而SingleCall则要等到调用对象方法时再创建。其次，SingleCall模式激活的对象是无状态的，对象声明周期由GC管理，而客户端激活的对象是有状态的，其生命周期可自定义。第三，两种激活模式在服务器端和客户端实现的方法不一样，尤其是在客户端，SingleCall模式由GetObject()来激活，它调用对象默认的构造函数，而客户端激活模式，则通过CreateInstance()来激活，它可以传递参数，所以可以调用自定义的构造函数来创建实例。

3. 通道：在.NET Remoting中时通过通道来实现两个应用程序域之间对象的通信。.NET Remoting中包括4中通道类型:

1. TcpChannel：Tcp通道使用Tcp协议来跨越.Net Remoting边界来传输序列化的消息流，TcpChannel默认使用二进制格式序列化消息对象，因此具有更高的传输性能，但不提供任何内置的安全功能。
2. HttpChannel：Http通道使用Http协议在客户端和服务器之间发生消息，使其在Internet上穿越防火墙来传输序列化的消息流（这里准确讲不能说穿越，主要是因为防火墙都开放了80端口，所以使用Http协议可以穿过防火墙进行数据的传输，如果防火墙限制了80端口，Http协议也照样不能穿越防火墙）。默认情况下，HttpChannel使用Soap格式序列化消息对象，因此它具有更好的互操作性，并且可以使用Http协议中的加密机制来对消息进行加密来保证安全性。因此，通常在局域网内，我们更多地使用TcpChannel，如果要穿越防火墙，则使用HttpChannel。
3. IpcChannel：进程间通信，只使用同一个系统进程之间的通信，不需要主机名和端口号。而使用Http通道和Tcp通道都要指定主机名和端口号。
4. 自定义通道：自定义的传输通道可以使用任何基本的传输协议来进行通信，如UDP协议、SMTP协议等。

三、利用.NET Remoting技术开发分布式应用三部曲

前面详细介绍了.NET Remoting相关内容，下面具体看看如何使用.NET Remoting技术来开发分布式应用程序。开发.NET Remoting应用分三步走。

第一步：创建远程对象，该对象必须继承MarshalByRefObject对象。具体的示例代码如下：

```
 1 namespace RemotingObject
 2 {
 3     // 第一步：创建远程对象
 4     // 创建远程对象——必须继承MarshalByRefObject,该类支持对象的跨域边界
 5     public class MyRemotingObject :MarshalByRefObject
 6     {
 7         // 用来测试Tcp通道
 8         public int AddForTcpTest(int a, int b)
 9         {
10             return a + b;
11         }
12 
13         // 用来测试Http通道
14         public int MinusForHttpTest(int a, int b)
15         {
16             return a - b;
17         }
18 
19         // 用来测试IPC通道
20         public int MultipleForIPCTest(int a, int b)
21         {
22             return a * b;
23         }
24     }
25 }
```

远程对象分别定义3个方法，目的是为了测试3中不同的通道方式的效果。

第二步：创建服务器端，需要添加System.Runtime.Remoting.dll引用，具体实现代码如下所示：

```
 1 using System;
 2 using System.Runtime.Remoting;
 3 using System.Runtime.Remoting.Channels;
 4 using System.Runtime.Remoting.Channels.Http;
 5 using System.Runtime.Remoting.Channels.Ipc;
 6 using System.Runtime.Remoting.Channels.Tcp;
 7
 8 namespace RemotingServerHost
 9 {
10     // 第二步：创建宿主应用程序
11     class Server
12     {
13         static void Main(string[] args)
14         {
15             // 1.创建三种通道
16
17             // 创建Tcp通道，端口号9001
18             TcpChannel tcpChannel = new TcpChannel(9001);
19
20             // 创建Http通道，端口号9002
21             HttpChannel httpChannel = new HttpChannel(9002);
22
23             // 创建IPC通道，端口号9003
24             IpcChannel ipcChannel = new IpcChannel("IpcTest");
25
26             // 2.注册通道
27             ChannelServices.RegisterChannel(tcpChannel, false);
28             ChannelServices.RegisterChannel(httpChannel, false);
29             ChannelServices.RegisterChannel(ipcChannel, false);
30
31             // 打印通道信息
32             // 打印Tcp通道的名称
33             Console.WriteLine("The name of the TcpChannel is {0}
34             // 打印Tcp通道的优先级
35             Console.WriteLine("The priority of the TcpChannel is
36
37             Console.WriteLine("The name of the HttpChannel is {0
38             Console.WriteLine("The priority of the httpChannel i
39
40             Console.WriteLine("The name of the IpcChannel is {0}
41             Console.WriteLine("The priority of the IpcChannel is
42
43             // 3\. 注册对象
44             // 注册MyRemotingObject到.NET Remoting运行库中
45             RemotingConfiguration.RegisterWellKnownServiceType(t
46             Console.WriteLine("Press any key to exit");
47             Console.ReadLine();
48         }
49     }
50 }
```

第三步：创建客户端程序，具体的实现代码如下所示：

```
 1 using RemotingObject;
 2 using System;
 3 
 4 namespace RemotingClient
 5 {
 6     class Client
 7     {
 8         static void Main(string[] args)
 9         {
10             // 使用Tcp通道得到远程对象
11             //TcpChannel tcpChannel = new TcpChannel();
12             //ChannelServices.RegisterChannel(tcpChannel, false)
13             MyRemotingObject proxyobj1 = Activator.GetObject(typ
14             if (proxyobj1 == null)
15             {
16                 Console.WriteLine("连接TCP服务器失败");
17             }
18 
19             //HttpChannel httpChannel = new HttpChannel();
20             //ChannelServices.RegisterChannel(httpChannel, false
21             MyRemotingObject proxyobj2 = Activator.GetObject(typ
22             if (proxyobj2 == null)
23             {
24                 Console.WriteLine("连接Http服务器失败");
25             }
26 
27             //IpcChannel ipcChannel = new IpcChannel();
28             //ChannelServices.RegisterChannel(ipcChannel, false)
29             MyRemotingObject proxyobj3 = Activator.GetObject(typ
30             if (proxyobj3 == null)
31             {
32                 Console.WriteLine("连接Ipc服务器失败");
33             }
34             // 输出信息
35             Console.WriteLine("This call object by TcpChannel, 1
36             Console.WriteLine("This call object by HttpChannel,
37             Console.WriteLine("This call object by IpcChannel, 1
38             Console.WriteLine("Press any key to exit!");
39             Console.ReadLine();
40         }
41     }
42 }
```

经过上面的三步，我们就完成了这个分布式应用的开发工作，下面测试下该程序是否可以正常运行，首先，运行服务器端，你将看到如下界面：

```
file:///F:/Study/C#/博客园中例子/深入理解WCF系列/NetRemoting/RemotingServerHost/bin/...
The name of the TcpChannel is tcp
The priority of the TcpChannel is 1
The name of the HttpChannel is http
The priority of the httpChannel is 1
The name of the IpcChannel is ipc
The priority of the IpcChannel is 20
Press any key to exit
```

在.NET Remoting中，是允许同时创建多个通道的，但是.NET Remoting要求通道的名字必须不同，因为名字是用来标识通道的唯一标识符。但上面代码中，我们并没有指明通道的名字，为什么还可以允许成功呢？从上面图片可知，当我们创建通道时，如果没有为其显式指定通道名，则会使用对应的通道类型作为该通道名，如TcpChannel将会以tcp作为通道名，如果想注册多个Tcp通道则必须显式指定其名字。

下面看看运行客户端所获得的结果，具体客户端运行效果如下图所示：

```
file:///F:/Study/C#/博客园中例子/深入理解WCF系列/NetRemoting/RemotingClient/bin/Deb...
This call object by TcpChannel, 100 + 200 = 300
This call object by HttpChannel, 100 - 200 = -100
This call object by IpcChannel, 100 * 200 = 20000
Press any key to exit!
```

四、使用配置文件来重写上面的分布式程序

在第三部分中，我们是把服务器的各种通道方式和地址写死在程序中的，这样的实现方式部署起来不方便，下面使用配置文件的方式来配置服务器端的通道类型和服务器地址。远程对象的定义不需要改变，下面直接看服务器端使用配置文件后的实现代码如下所示：

```
1 using System;
2 using System.Runtime.Remoting;
3 using System.Runtime.Remoting.Channels;
4
5 namespace RemotingServerHostByConfig
6 {
7     class Program
8     {
9         static void Main(string[] args)
10        {
11            RemotingConfiguration.Configure("RemotingServerHostE
12
13            foreach (var channel in ChannelServices.RegisteredCh
14            {
15                // 打印通道的名称
16                Console.WriteLine("The name of the Channel is {(
17                // 打印通道的优先级
18                Console.WriteLine("The priority of the Channel i
19            }
20            Console.WriteLine("按任意键退出……");
21            Console.ReadLine();
22        }
23    }
24 }
```

服务端的配置文件的内容为:

```
1 <?xml version="1.0" encoding="utf-8" ?>
2 <!--服务端App.config的内容-->
3 <configuration>
4     <startup>
5         <supportedRuntime version="v4.0" sku=".NETFramework,Vers
6     </startup>
7   <system.runtime.remoting>
8     <application>
9       <service>
10        <wellknown  mode="Singleton"
11                    type="RemotingObject.MyRemotingObject,Remoti
12                    objectUri="MyRemotingObject"/>
13      </service>
14      <channels>
15        <channel port="9001" ref="tcp"/>
16        <channel port="9002" ref="http"/>
17        <channel portName="IpcTest" ref="ipc"/> <!--Ipc通道不需要
18      </channels>
19    </application>
20  </system.runtime.remoting>
21 </configuration>
```

此时，客户端程序的实现代码如下所示：

```
 1 using RemotingObject;
 2 using System;
 3 using System.Runtime.Remoting;
 4 
 5 namespace RemotingClientByConfig
 6 {
 7     class Program
 8     {
 9         static void Main(string[] args)
10         {
11             //使用HTTP通道得到远程对象
12             RemotingConfiguration.Configure("RemotingClientByCon
13             MyRemotingObject proxyobj1 = new MyRemotingObject();
14             if (proxyobj1 == null)
15             {
16                 Console.WriteLine("连接服务器失败");
17             }
18 
19             Console.WriteLine("This call object by TcpChannel, 1
20             Console.WriteLine("This call object by HttpChannel, 
21             Console.WriteLine("This call object by IpcChannel, 1
22             Console.WriteLine("Press any key to exit!");
23             Console.ReadLine();
24         }
25     }
26 }
```

客户端配置文件为：

```
1 <?xml version="1.0" encoding="utf-8" ?>
2 <configuration>
3     <startup>
4         <supportedRuntime version="v4.0" sku=".NETFramework,Vers
5     </startup>
6   <system.runtime.remoting>
7     <application>
8       <client>
9         <wellknown  type="RemotingObject.MyRemotingObject,Remot
10                     url="http://localhost:9002/MyRemotingObject'
11       </client>
12       <channels>
13         <channel ref="tcp" port="0"></channel>
14         <channel ref="http" port="0"></channel>
15         <channel ref="ipc" port="0"></channel>
16       </channels>
17     </application>
18   </system.runtime.remoting>
19 </configuration>
```

使用配置文件修改后的分布式程序的运行结果与前面的运行结果一样，这里就不一一贴图了。

五、总结

到这里，.NET Remoting技术的分享就结束了，本文只是对.NET Remoting技术做了一个基本的介绍，如果想深入了解.NET Remoting技术的话，推荐大家可以看看下面的专题细细品味C#——.Net Remoting专题。在下一篇文章中，继续为大家分享另一种分布式技术——Web Service。

本文的示例代码文件下载：.NETRemotingSample

跟我一起学WCF(3)——利用Web Services开发分布式应用

一、引言

在前面文章中分别介绍了MSMQ和.NET Remoting技术，今天继续分享.NET 平台下另一种分布式技术——Web Services

二、Web Services 详细介绍

2.1 Web Services 概述

Web Services是支持客户端与服务器通过网络互操作的一种软件系统，是一组可以通过网络调用的应用程序API。在Web Services中主要到SOAP/UDDI/WSDL这三个核心概念，下面分别介绍下这三个概念的定义。

- SOAP：SOAP（Simple Object Access Protocol，简单对象访问协议）是在分散或分布式的环境中交换信息的简单协议，是一种基于XML的协议，需要绑定一个网络传输协议来完成信息的传输，这个协议通常是Http或Https，但也可以使其他协议。

它包括四个部分：

SOAP封装：它定义了一个框架，描述消息中的内容是描述，是谁发送的，谁又应当接收并处理；

SOAP编码规则：定义了一种序列化的机制，用于表示应用程序需要使用的数据类型的实例；

SOAP RPC：表示一种协定，用于表示远程过程调用和应答；

SOAP绑定：它定义了SOAP使用哪种协议来进行交换信息。使用Http/TCP/UDP都可以。与WCF中的绑定概念一致。

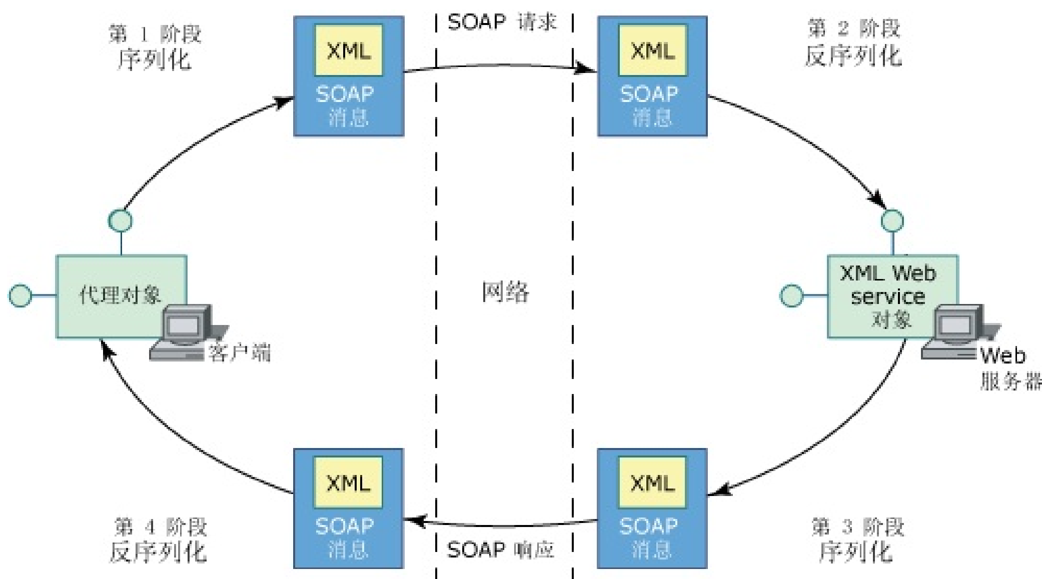
换句话说，SOAP协议只是用来封装消息用的，封装后的消息你可以通过各种已有的协议来传输，如Http、Https、Tcp、UDP、SMTP等，甚至你还可以自定义协议。然而Web Service是采用基于Http协议来传输数据的。关于使用Https协议来访问Web Services的方法可以参考这个文章：如何利用 SSL 调用 Web 服务。

- UDDI：是统一描述、发现和集成（Universal Description, Discovery, and Integration）的缩写，它是一个基于XML的跨平台的描述规范，可以使世界范围内的企业在互联网上发布自己所提供的服务供其他客户查询使用。
- WSDL：是Web服务描述语言（Web Services Description Language），是为描述Web服务发布的XML格式。用于描述服务器端口访问方式和使用协议的细

节，通常用来辅助生产服务器和客户端代码及配置信息。

2.2 Web Services 实现过程

调用Web Services的实现过程与进行常规方法调用过程类似。不同的在于，前者方法并不位于客户端应用程序中，而是通过指定传输协议生成请求消息。因为Web Services可能位于不同的计算机上，因此必须将Web Services处理请求所需的信息通过网络传递给含有Web Services的服务器，Web Services在处理信息后，会通过网络将结果发送回客户端应用程序。下图显示了客户端与Web Services之间的通信过程：



下面介绍下调用Web Services时事件发生顺序：

1. 在客户端上，创建了一个Web Services代理类的实例。该对象驻留在客户端机器上。
2. 客户端调用代理类上的方法
3. 客户端机器将Web Services方法的参数序列化为SOAP消息，然后通过传送协议发送给Web Services。
4. Web Services底层结构接收SOAP消息并进行反序列化。它会创建Web Services的类的实例，同时调用对应的Web Services方法。
5. Web Services方法执行，并返回结果。
6. Web Services底层结构会将返回结果序列化为SOAP消息，然后通过网络发送回客户端。
7. 客户端将接收SOAP消息，然后将XML反序列为返回值或任何输出参数，并将它们传递给代理类的实例。
8. 客户端接收返回值和所有输出参数。

2.3 Web Services 优缺点

经过上面详细的介绍后，Web Services很明显具有以下优点：

- 跨平台：Web Services完全基于XML（可扩展标记语言）、

XSD（XMLSchema）等与平台无关的行业标准。
- 自描述：Web Service使用WSDL进行自我描述，包括服务的方法、参数、类型和返回值等相关信息。
- 跨防火墙：Web Service使用http协议进行通信，可以穿越防火墙。

Web Services也具有以下缺点：

- 效率低下，不适合做单应用系统的开发。
- 安全问题：Web Services没有自身的安全机制，必须借助Http协议或IIS等宿主程序实现信息安全加密。

三、使用**Web Services**来开发分布式应用程序

使用Web Services来开发分布式应用较MSMQ和.NET Remoting来说相对简单很多，今天的示例程序分三步走：

1. 创建一个实现用户信息验证的项目WebServiceUserValidation。具体的实现代码如下所示：

```
 1 namespace WebServiceUserValidation
 2 {
 3     public class UserValidation
 4     {
 5         // 判断用户名和密码是否有效
 6         public static bool IsUserLegal(string name, string psw)
 7         {
 8             // 用户可以访问数据库进行用户和密码验证
 9             // 这里仅仅作为演示
10             string password = "LearningHard";
11             if (string.Equals(password, psw))
12             {
13                 return true;
14             }
15             else
16             {
17                 return false;
18             }
19         }
20 
21         // 判断用户的凭证是否有效
22         public static bool IsUserLegal(string token)
23         {
24             // 用户可以访问数据库进行用户凭证验证
25             // 这里只做演示
26             string password = "LearningHard";
27             if (string.Equals(password, token))
28             {
29                 return true;
30             }
31             else
32             {
33                 return false;
34             }
35         }
36     }
37 }
```

2. 创建Web Services服务类，需要创建一个继承自SoapHeader.aspx)，来接收SOAP 头里的消息，并添加WebServiceUserValidation程序集。通过添加Asp.net空Web应用程序来创建Web Services服务工程，再右键创建的Web 应用程序工程添加一个Web 服务文件来创建Web 服务。具体的实现代码如下所示：

```
1 // 用户自定义的SoapHeader类必须继承于SoapHeader
2     public class MySoapHeader : SoapHeader
3     {
4         // 存储用户凭证
5         public string Token { get; set; }
6     }
7  /// <summary>
8     /// LearningHardWebService 的摘要说明
9     /// </summary>
10     [WebService(Namespace = "http://www.cnblogs.com/zhili/")]
11     [WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1
12     [System.ComponentModel.ToolboxItem(false)]
13     // 若要允许使用 ASP.NET AJAX 从脚本中调用此 Web 服务，请取消注释以
14     // [System.Web.Script.Services.ScriptService]
15     public class LearningHardWebService : System.Web.Services.We
16     {
17         // 存储用户凭证的Soap Header信息
18         // 必须保证是public和字段名必须与SoapHeader("memberName")中m
19         // 否则会出现“头属性/字段 LearningHardWebService.authentica
20         public MySoapHeader authenticationToken;
21         private const string TOKEN = "LearningHard"; // 存储服务器
22 
23 
24         // 定义SoapHeader传递的方向
25         //SoapHeaderDirection.In;只发送SoapHeader到服务端,该值是默认
26         //SoapHeaderDirection.Out;只发送SoapHeader到客户端
27         //SoapHeaderDirection.InOut;发送SoapHeader到服务端和客户端
28         //SoapHeaderDirection.Fault;服务端方法异常的话，会发送异常信息
29         [SoapHeader("authenticationToken", Direction = SoapHeade
30         [WebMethod(EnableSession = false)]
31         public string HelloLearningHard()
32         {
33             if (authenticationToken != null && UserValidation.Is
34             {
35                 return "LearningHard 你好，调用服务方法成功!";
36             }
37             else
38             {
39                 throw new SoapException("身份验证失败", SoapExcept
40             }
41         }
42     }
```

在上面代码中需要注意的是，Web Servies中的Web方法需要抛出SoapExcetion异常才能被客户端捕获到，如果在Debug模式下调试运行的话，还需要在异常设置里把这个异常勾选掉，即编译器不对该异常进行捕获。

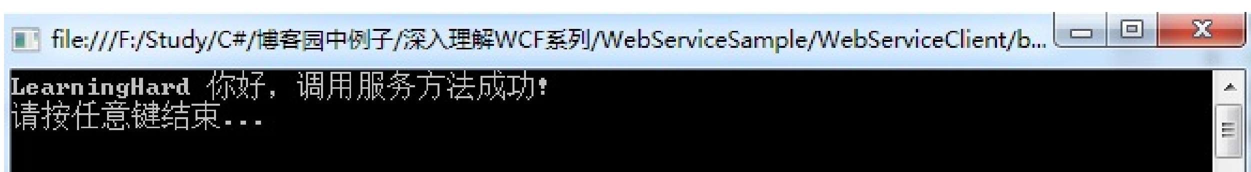
3. 创建控制台客户端，通过添加服务引用的方式来添加Web Services，添加成功后，会在客户端程序中创建一个代理类，客户端可以通过该代理类来调用Web Services的方法，具体的实现代码如下所示:

```
 1 namespace WebServiceClient
 2 {
 3     class Program
 4     {
 5         static void Main(string[] args)
 6         {
 7             // 实例化一个Soap协议的头
 8             MySoapHeader mySoapHeader = new MySoapHeader() { Tol
 9             string sResult = string.Empty;
10             LearningHardWebServiceSoapClient learningHardWebSer
11             try
12             {
13                 // 实例化Web服务的客户端代理类
14                 learningHardWebSer = new LearningHardWebServiceS
15                 // 调用Web服务上的方法
16                 sResult= learningHardWebSer.HelloLearningHard(re
17                 // 输出结果
18                 Console.WriteLine(sResult);
19             }
20             catch
21             {
22                 Console.WriteLine("调用Web服务失败!");
23             }
24             finally
25             {
26                 // 释放托管资源
27                 if (learningHardWebSer != null)
28                 {
29                     learningHardWebSer.Close();
30                 }
31             }
32 
33             Console.WriteLine("请按任意键结束...");
34             Console.ReadLine();
35         }
36     }
37 }
```

关于Web Services异常捕获的更多信息可以参考MSDN：在 XML Web services 中处理和引发异常.aspx)。然而在这个MSDN上的示例代码好像运行不成功，后面发现，该文章中的客户端对异常的处理只处理了SoapException 异常，而此时客户端触发的异常时FaultException异常，所以异常处理代码应像下面代码一样处理，当然也可以直接只处理Exception异常，我上面代码就只处理这个大范围的异常。

```
catch (SoapException ex)
    {
        // Do sth with SoapException
    }
    catch (Exception ex)
    {
        // Do sth with Exception
    }
```

经过上面的步骤，我们就已经完成了所有的开发工作，下面运行来测试下该程序的运行效果。把WebServiceClient作为启动项目，直接按F5或Ctrl+F5来运行客户端程序，你将看到如下所示的结果：



注：像一般分布式应用程序，都应用先运行服务器端，再运行客户端来访问服务方法。而这里我们运行却直接运行客户端就可以访问Web Services中的Web方法了。这是因为在运行Web Services客户端程序之前，会先把Web Services部署到IIS Express 中，你将会看到任务栏右下角有 ，右键该图标就可以看到运行的Web Services。

四、总结

到这里，Web Services技术的分享就结束，从下一篇文章开始，将正式进入WCF的世界。而Web Services的内容和WCF内容一样也有很多，只是微软官方推荐采用WCF来创建Web服务程序，如果你想更多地了解Web Services的内容，可以参考MSDN：使用 ASP.NET 创建的 XML Web Services 以及 XML Web Services 客户端.aspx)

本文所有示例代码下载：WebServiceSample

跟我一起学WCF(3)——利用Web Services开发分布式应用

一、引言

在前面文章中分别介绍了MSMQ和.NET Remoting技术，今天继续分享.NET 平台下另一种分布式技术——Web Services

二、Web Services 详细介绍

2.1 Web Services 概述

Web Services是支持客户端与服务器通过网络互操作的一种软件系统，是一组可以通过网络调用的应用程序API。在Web Services中主要到SOAP/UDDI/WSDL这三个核心概念，下面分别介绍下这三个概念的定义。

- SOAP：SOAP（Simple Object Access Protocol，简单对象访问协议）是在分散或分布式的环境中交换信息的简单协议，是一种基于XML的协议，需要绑定一个网络传输协议来完成信息的传输，这个协议通常是Http或Https，但也可以使其他协议。

它包括四个部分：

SOAP封装：它定义了一个框架，描述消息中的内容是描述，是谁发送的，谁又应当接收并处理；

SOAP编码规则：定义了一种序列化的机制，用于表示应用程序需要使用的数据类型的实例；

SOAP RPC：表示一种协定，用于表示远程过程调用和应答；

SOAP绑定：它定义了SOAP使用哪种协议来进行交换信息。使用Http/TCP/UDP都可以。与WCF中的绑定概念一致。

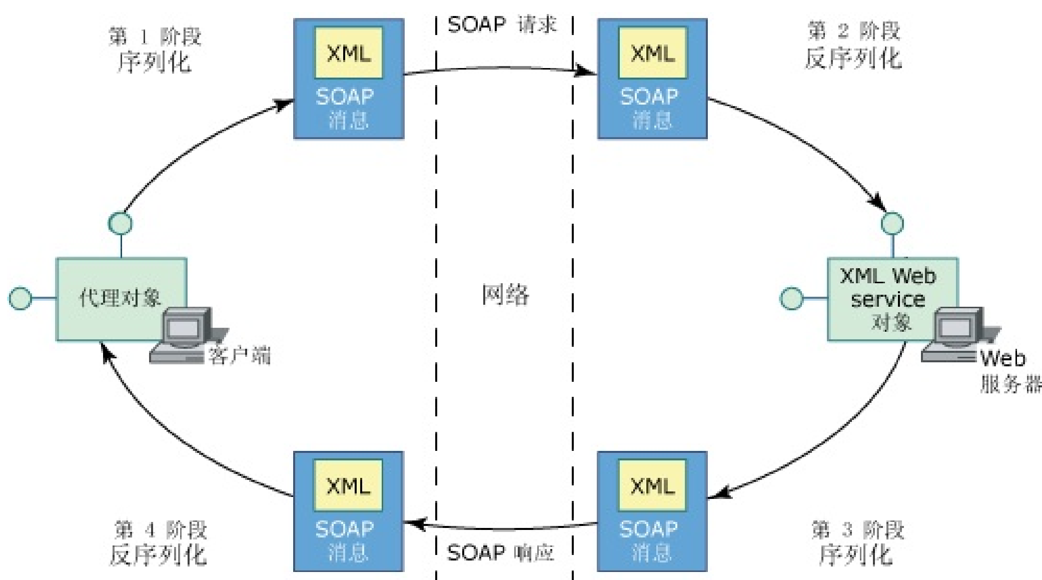
换句话说，SOAP协议只是用来封装消息用的，封装后的消息你可以通过各种已有的协议来传输，如Http、Https、Tcp、UDP、SMTP等，甚至你还可以自定义协议。然而Web Service是采用基于Http协议来传输数据的。关于使用Https协议来访问Web Services的方法可以参考这个文章：如何利用 SSL 调用 Web 服务。

- UDDI：是统一描述、发现和集成（Universal Description, Discovery, and Integration）的缩写，它是一个基于XML的跨平台的描述规范，可以使世界范围内的企业在互联网上发布自己所提供的服务供其他客户查询使用。
- WSDL：是Web服务描述语言（Web Services Description Language），是为描述Web服务发布的XML格式。用于描述服务器端口访问方式和使用协议的细

节，通常用来辅助生产服务器和客户端代码及配置信息。

2.2 Web Services 实现过程

调用Web Services的实现过程与进行常规方法调用过程类似。不同的在于，前者方法并不位于客户端应用程序中，而是通过指定传输协议生成请求消息。因为Web Services可能位于不同的计算机上，因此必须将Web Services处理请求所需的信息通过网络传递给含有Web Services的服务器，Web Services在处理信息后，会通过网络将结果发送回客户端应用程序。下图显示了客户端与Web Services之间的通信过程：



下面介绍下调用Web Services时事件发生顺序：

1. 在客户端上，创建了一个Web Services代理类的实例。该对象驻留在客户端机器上。
2. 客户端调用代理类上的方法
3. 客户端机器将Web Services方法的参数序列化为SOAP消息，然后通过传送协议发送给Web Services。
4. Web Services底层结构接收SOAP消息并进行反序列化。它会创建Web Services的类的实例，同时调用对应的Web Services方法。
5. Web Services方法执行，并返回结果。
6. Web Services底层结构会将返回结果序列化为SOAP消息，然后通过网络发送回客户端。
7. 客户端将接收SOAP消息，然后将XML反序列为返回值或任何输出参数，并将它们传递给代理类的实例。
8. 客户端接收返回值和所有输出参数。

2.3 Web Services 优缺点

经过上面详细的介绍后，Web Services很明显具有以下优点：

- 跨平台：Web Services完全基于XML（可扩展标记语言）、

XSD（XMLSchema）等与平台无关的行业标准。
- 自描述：Web Service使用WSDL进行自我描述，包括服务的方法、参数、类型和返回值等相关信息。
- 跨防火墙：Web Service使用http协议进行通信，可以穿越防火墙。

Web Services也具有以下缺点：

- 效率低下，不适合做单应用系统的开发。
- 安全问题：Web Services没有自身的安全机制，必须借助Http协议或IIS等宿主程序实现信息安全加密。

三、使用**Web Services**来开发分布式应用程序

使用Web Services来开发分布式应用较MSMQ和.NET Remoting来说相对简单很多，今天的示例程序分三步走：

1. 创建一个实现用户信息验证的项目WebServiceUserValidation。具体的实现代码如下所示：

```
 1 namespace WebServiceUserValidation
 2 {
 3     public class UserValidation
 4     {
 5         // 判断用户名和密码是否有效
 6         public static bool IsUserLegal(string name, string psw)
 7         {
 8             // 用户可以访问数据库进行用户和密码验证
 9             // 这里仅仅作为演示
10             string password = "LearningHard";
11             if (string.Equals(password, psw))
12             {
13                 return true;
14             }
15             else
16             {
17                 return false;
18             }
19         }
20 
21         // 判断用户的凭证是否有效
22         public static bool IsUserLegal(string token)
23         {
24             // 用户可以访问数据库进行用户凭证验证
25             // 这里只做演示
26             string password = "LearningHard";
27             if (string.Equals(password, token))
28             {
29                 return true;
30             }
31             else
32             {
33                 return false;
34             }
35         }
36     }
37 }
```

2. 创建Web Services服务类，需要创建一个继承自SoapHeader.aspx)，来接收SOAP 头里的消息，并添加WebServiceUserValidation程序集。通过添加Asp.net空Web应用程序来创建Web Services服务工程，再右键创建的Web 应用程序工程添加一个Web 服务文件来创建Web 服务。具体的实现代码如下所示：

```
1 // 用户自定义的SoapHeader类必须继承于SoapHeader
2     public class MySoapHeader : SoapHeader
3     {
4         // 存储用户凭证
5         public string Token { get; set; }
6     }
7  /// <summary>
8     /// LearningHardWebService 的摘要说明
9     /// </summary>
10    [WebService(Namespace = "http://www.cnblogs.com/zhili/")]
11    [WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)
12    [System.ComponentModel.ToolboxItem(false)]
13    // 若要允许使用 ASP.NET AJAX 从脚本中调用此 Web 服务，请取消注释以下
14    // [System.Web.Script.Services.ScriptService]
15    public class LearningHardWebService : System.Web.Services.We
16    {
17        // 存储用户凭证的Soap Header信息
18        // 必须保证是public和字段名必须与SoapHeader("memberName")中m
19        // 否则会出现“头属性/字段 LearningHardWebService.authentica
20        public MySoapHeader authenticationToken;
21        private const string TOKEN = "LearningHard"; // 存储服务器
22
23
24        // 定义SoapHeader传递的方向
25        //SoapHeaderDirection.In;只发送SoapHeader到服务端,该值是默认
26        //SoapHeaderDirection.Out;只发送SoapHeader到客户端
27        //SoapHeaderDirection.InOut;发送SoapHeader到服务端和客户端
28        //SoapHeaderDirection.Fault;服务端方法异常的话，会发送异常信息
29        [SoapHeader("authenticationToken", Direction = SoapHeade
30        [WebMethod(EnableSession = false)]
31        public string HelloLearningHard()
32        {
33            if (authenticationToken != null && UserValidation.Is
34            {
35                return "LearningHard 你好，调用服务方法成功!";
36            }
37            else
38            {
39                throw new SoapException("身份验证失败", SoapExcept
40            }
41        }
42    }
```

在上面代码中需要注意的是，Web Servies中的Web方法需要抛出SoapExcetion异常才能被客户端捕获到，如果在Debug模式下调试运行的话，还需要在异常设置里把这个异常勾选掉，即编译器不对该异常进行捕获。

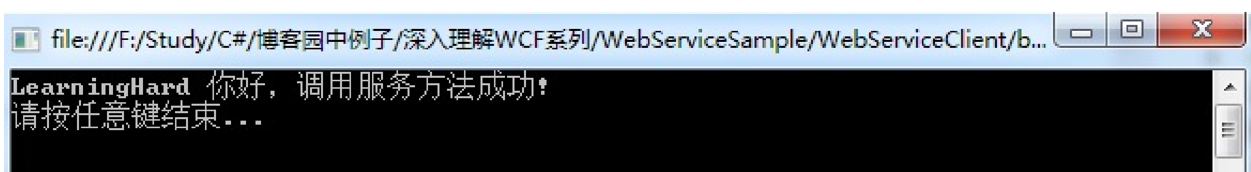
3. 创建控制台客户端，通过添加服务引用的方式来添加Web Services，添加成功后，会在客户端程序中创建一个代理类，客户端可以通过该代理类来调用Web Services的方法，具体的实现代码如下所示:

```
 1 namespace WebServiceClient
 2 {
 3     class Program
 4     {
 5         static void Main(string[] args)
 6         {
 7             // 实例化一个Soap协议的头
 8             MySoapHeader mySoapHeader = new MySoapHeader() { Tol
 9             string sResult = string.Empty;
10             LearningHardWebServiceSoapClient learningHardWebSer
11             try
12             {
13                 // 实例化Web服务的客户端代理类
14                 learningHardWebSer = new LearningHardWebServiceS
15                 // 调用Web服务上的方法
16                 sResult= learningHardWebSer.HelloLearningHard(re
17                 // 输出结果
18                 Console.WriteLine(sResult);
19             }
20             catch
21             {
22                 Console.WriteLine("调用Web服务失败!");
23             }
24             finally
25             {
26                 // 释放托管资源
27                 if (learningHardWebSer != null)
28                 {
29                     learningHardWebSer.Close();
30                 }
31             }
32 
33             Console.WriteLine("请按任意键结束...");
34             Console.ReadLine();
35         }
36     }
37 }
```

关于Web Services异常捕获的更多信息可以参考MSDN：在 XML Web services 中处理和引发异常.aspx)。然而在这个MSDN上的示例代码好像运行不成功，后面发现，该文章中的客户端对异常的处理只处理了SoapException 异常，而此时客户端触发的异常时FaultException异常，所以异常处理代码应像下面代码一样处理，当然也可以直接只处理Exception异常，我上面代码就只处理这个大范围的异常。

```
catch (SoapException ex)
    {
        // Do sth with SoapException
    }
    catch (Exception ex)
    {
        // Do sth with Exception
    }
```

经过上面的步骤，我们就已经完成了所有的开发工作，下面运行来测试下该程序的运行效果。把WebServiceClient作为启动项目，直接按F5或Ctrl+F5来运行客户端程序，你将看到如下所示的结果：



注：像一般分布式应用程序，都应用先运行服务器端，再运行客户端来访问服务方法。而这里我们运行却直接运行客户端就可以访问Web Services中的Web方法了。这是因为在运行Web Services客户端程序之前，会先把Web Services部署到IIS Express 中，你将会看到任务栏右下角有 ，右键该图标就可以看到运行的Web Services。

四、总结

到这里，Web Services技术的分享就结束，从下一篇文章开始，将正式进入WCF的世界。而Web Services的内容和WCF内容一样也有很多，只是微软官方推荐采用WCF来创建Web服务程序，如果你想更多地了解Web Services的内容，可以参考MSDN：使用 ASP.NET 创建的 XML Web Services 以及 XML Web Services 客户端.aspx)

本文所有示例代码下载：WebServiceSample

跟我一起学WCF(4)——第一个WCF程序

一、引言

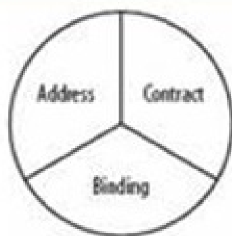
前面几篇文章分享了.NET 平台下其他几种分布式技术，然而前面几种分布式技术专注于某一特定的领域，并且具有不同编程接口，这使得开发人员需要掌握多个API的使用。基于这样的原因，微软在.NET 3.0时实现了WCF。WCF是.NET平台下各种分布式技术的集成，它将前面介绍的几种分布式技术完全整合在一起，并提供了一套统一的编程接口（API）。对于，开发人员来来说只需要掌握WCF一套的API，就可以实现之前分布式技术所实现的所有功能。

二、WCF详细介绍

WCF（Windows Communication Foundation）是微软为构建面向服务的应用程序（SOA）而提供的统一编程模型，借助该模型，使得在构建分布式系统中，无需再考虑如何去实现通信的相关的问题，使开发人员更加关注与系统业务逻辑本身的实现。而在WCF中，各个Application之间的通信是由Endpoint(终结点)来实现的。下面详细介绍下WCF几个重要的概念。

2.1 EndPoint 详细介绍

服务的提供者将服务通过一个或多个终结点发布给潜在的服务消费者，服务的消费者则通过与之匹配的终结点对服务进行消费。终结点由地址（Address）、绑定（Binding）和契约（Contract）三要素组成。如下图所示。由于它们的首字母分别是A、B、C。所以就有了：EndPoint = ABC。



这三个要素在WCF通信中起到的作用分别是：

- 地址（**Address**）：地址标识了服务的位置，提供寻址的辅助信息和标识了服务的真实身份。Address解决了**Where the WCF service?**的问题。
- 绑定（**Binding**）：绑定实现了通信的所有细节，包括网络传输，消息编码，以及其他为实现某种功能对消息进行的相应处理，例如安全、可靠传输和事务等功能。 WCF中具有一系列的系统已定义的绑定，如BasicHttpBinding、WsHttpBinding、NetTcpBinding等。Binding解决了How to Communicate with Service？的问题。

- 契约（**Contract**）：契约是对服务操作的抽象，也是对消息交互模式以及消息结构的定义。WCF的契约大体可以分为两类，一类是对服务操作的描述；另一类是对数据的描述。服务契约(Service Contract)则属于对服务操作的描述，而后一类包括其余3中契约：数据契约（Data Contract）、消息契约（Message Contract）和错误契约（Fault Contract）。Contract解决了What function does the Service Provide?的问题。

2.2 WCF 基础概念

- 消息模式

消息时一个独立的数据单元，它可能由几个部分组成，包括消息正文和消息头。WCF支持多种消息模式，包括请求-恢复、单向和双工通信。不同传输协议支持不同的消息模式，WCF API和运行库还能保证安全而可靠地发送消息

- 通信协议和编码

WCF支持Http、TCP、Peer network(对等网)、IPC(基于命名管道的内部进程通信)和MSMQ协议。在进行消息传递之前，必须对给定的消息进行格式化的编码，WCF提供了3种编码选择：一种是文本编码，一种跨平台的编码；一种是消息传输优化机制（MOMO）编码，该编码用于高效地将非结构化的二进制数据发送到服务或从服务接收这些数据。一种是用于实现高效传输的二进制编码。

- 行为（Behavior）：Behavior的主要作用是定制EndPoint在运行时的一些不要的行为。比如Service回调Client的TimeOut属性的设置。
- 宿主和宿主进程：服务必须承载于某个进程中，宿主进程是专为承载服务而设计的应用程序，这些宿主进程包括Internet信息服务（即IIS）、Windows激活服务（WAS）、Windows服务。由宿主控制服务的生命周期。
- 自承载服务：服务除了可以由现有的宿主进程承载外，还可以自承载，自承载服务是由开发人员创建的进程应用程序来承载服务。该应用程序控制服务的声明周期，设置服务的属性和打开服务和关闭服务等操作。

三、创建第一个**WCF**应用程序

前面介绍了WCF的详细内容，接下来，我们采用以下两种服务寄宿方法来创建WCF应用程序。

- 通过自我寄宿（Self-Hosting）的方式，即自承载服务。创建一个控制台应用程序来作为服务的宿主。
- 通过IIS寄宿方式将服务寄宿在IIS中。客户端通过另一个控制台程序来模拟客户端来对服务进行调用。

接下来，我们一步一步来使用两种方式来实现我们的WCF应用程序。首先介绍自我寄宿方式的实现步骤。

步骤一：创建服务契约和服务

既然是分布式应用程序，首先第一步肯定是创建供其他消费者消费的服务应用程序。而WCF采用基于契约的交互方式实现了服务的自治，以及客户端和服务端之间的松耦合。从功能角度上，服务契约抽象了服务提供的所有操作，所以，我们一般通过接口的形式定义服务契约。WCF通过在接口上应用System.ServiceModel.Service.ServiceContractAttribute.aspx)特性将一个接口定义成服务契约。在应用该特性的同时，还可以指定服务契约的名称和命名空间，在这个服务契约中，我们将契约名称和命名空间设置成HellworldService和http://www.Learninghard.com 。应用ServiceContractAttribute特性将接口定义成服务契约之后，接口中定义的方法也不能自动成为服务的操作方法。此时，我们需要在相应的方法上显式地应用OperationContractAttribute.aspx)特性。具体服务契约的实现代码如下所示：

服务契约成功创建之后，我们需要实现服务契约来创建具体的WCF服务。WCF服务的具体实现代码如下所示：

```
1  public class HelloWorldService : IHelloWorld
2      {
3          public string GetHelloWorld()
4          {
5              return "Hello World";
6          }
7      }
```

步骤二：创建服务宿主

前面介绍到，WCF服务必须寄存在某个进程中，该进程称为宿主应用程序。服务寄宿的目的就是开启一个进程来为WCF服务提供一个运行的环境。通过为服务添加一个或多个终结点，使之暴露给服务消费者使用。服务消费者再通过相应匹配的终结点对服务进行调用，下面通过创建一个控制台程序来实现WCF服务的自我寄宿方式，具体的实现代码如下所示：

```
 1 using Contract;
 2 using Services;
 3 using System;
 4 using System.ServiceModel;
 5 using System.ServiceModel.Description;
 6 
 7 namespace Hosting
 8 {
 9     class Program
10     {
11         static void Main(string[] args)
12         {
13             using (ServiceHost host = new ServiceHost(typeof(He
14             {
15                 // 如果采用配置文件的方式，Region中代码就可以注释点
16                 #region
17                 host.AddServiceEndpoint(typeof(IHelloWorld), new
18                 if (host.Description.Behaviors.Find<ServiceMetad
19                 {
20                     ServiceMetadataBehavior behavior = new Servi
21                     behavior.HttpGetEnabled = true;
22                     behavior.HttpGetUrl = new Uri("http://127.0.
23                     host.Description.Behaviors.Add(behavior);
24                 }
25                 #endregion
26 
27                 host.Opened += delegate
28                 {
29                     Console.WriteLine("HelloWorldService 已经启动
30                 };
31 
32                 host.Open();
33                 Console.Read();
34             }
35         }
36     }
37 }
```

WCF服务寄宿通过ServiceHost.aspx)对象来完成的。在上面代码中，WCF实例是通过基于WCF服务的类型（typeof(HelloWorldService)）来创建的，并通过AddServiceEndpoint添加了一个终结点，具体终结点的地址为http://127.0.0.1:8888/HelloWorldService，采用的绑定（Binding）类型为WSHttpBinding，并指定了服务契约的类型为IHelloWorld。

WCF是SOA的实现，而SOA的一个基本特征是松耦合，WCF应用中客户端和服务端的松耦合体现在客户端只需了解WCF服务基本的描述，而无需知道具体的实现细节就可以实现对服务的访问。WCF服务的描述通过元数据的形式进行发布的。而WCF元数据的发布是通过一个服务行为ServiceMetadataBehavior.aspx)来实现的。在上面代码中，我们为创建的ServiceHost对象添加了

ServiceMetadataBehavior，并采用了基于Http-GET的元数据获取方式，元数据的发布地址指定为http://127.0.0.1:8888/HelloWorldService/metadata。在调用ServiceHost的Open方法对服务成功寄宿后，你可以通过该地址获取服务相关的元数据，就如Web 服务中通过输入WSDL地址来获得Web服务的描述一样。当我们成功运行ConsoleAppHosting宿主应用程序后，在浏览器中输入http://127.0.0.1:8888/HelloWorldService/metadata这个地址，你将得到如下所示的服务元数据。

This XML file does not appear to have any style information associated with it. The document tree is shown below.


```
▼<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/" xmlns:wsu="http://docs.oasis-
 open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/" xmlns:tns="http://tempuri.org/"
 xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
 xmlns:wsx="http://schemas.xmlsoap.org/ws/2004/09/mex"
 xmlns:wsap="http://schemas.xmlsoap.org/ws/2004/08/addressing/policy"
 xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl"
 xmlns:msc="http://schemas.microsoft.com/ws/2005/12/wsdl/contract"
 xmlns:i0="http://www.Learninghard.com" xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
 xmlns:wsa10="http://www.w3.org/2005/08/addressing"
 xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata" name="HelloWorldService"
 targetNamespace="http://tempuri.org/">
  ▼<wsp:Policy wsu:Id="WSHttpBinding_HellworldService_policy">
    ▼<wsp:ExactlyOne>
      ▼<wsp:All>
        ▼<sp:SymmetricBinding xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
          ▼<wsp:Policy>
            ▼<sp:ProtectionToken>
              ▼<wsp:Policy>
                ▼<sp:SecureConversationToken
                 sp:IncludeToken="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy/IncludeToke
                  ▼<wsp:Policy>
                     <sp:RequireDerivedKeys/>
                    ▼<sp:BootstrapPolicy>
                      ▼<wsp:Policy>
                        ▼<sp:SignedParts>
                           <sp:Body/>
                           <sp:Header Name="To" Namespace="http://www.w3.org/2005/08/addressing"/>
                           <sp:Header Name="From"
                           Namespace="http://www.w3.org/2005/08/addressing"/>
                           <sp:Header Name="FaultTo"
                           Namespace="http://www.w3.org/2005/08/addressing"/>
```


在运行宿主应用程序时，如果你没有以管理员权限运行宿主应用程序的话，你将会得到Http无法注册的异常，具体异常信息如下图所示：



此时，如果你是在VS中运行宿主程序，你就需要以管理员权限运行VS2012，如果直接在文件夹中运行宿主程序，此时需要右键宿主程序的exe文件，选择以管理员身份运行就不会出现如上图所示的异常。MSDN详细解释连接为：WCF福门教程疑难解答。

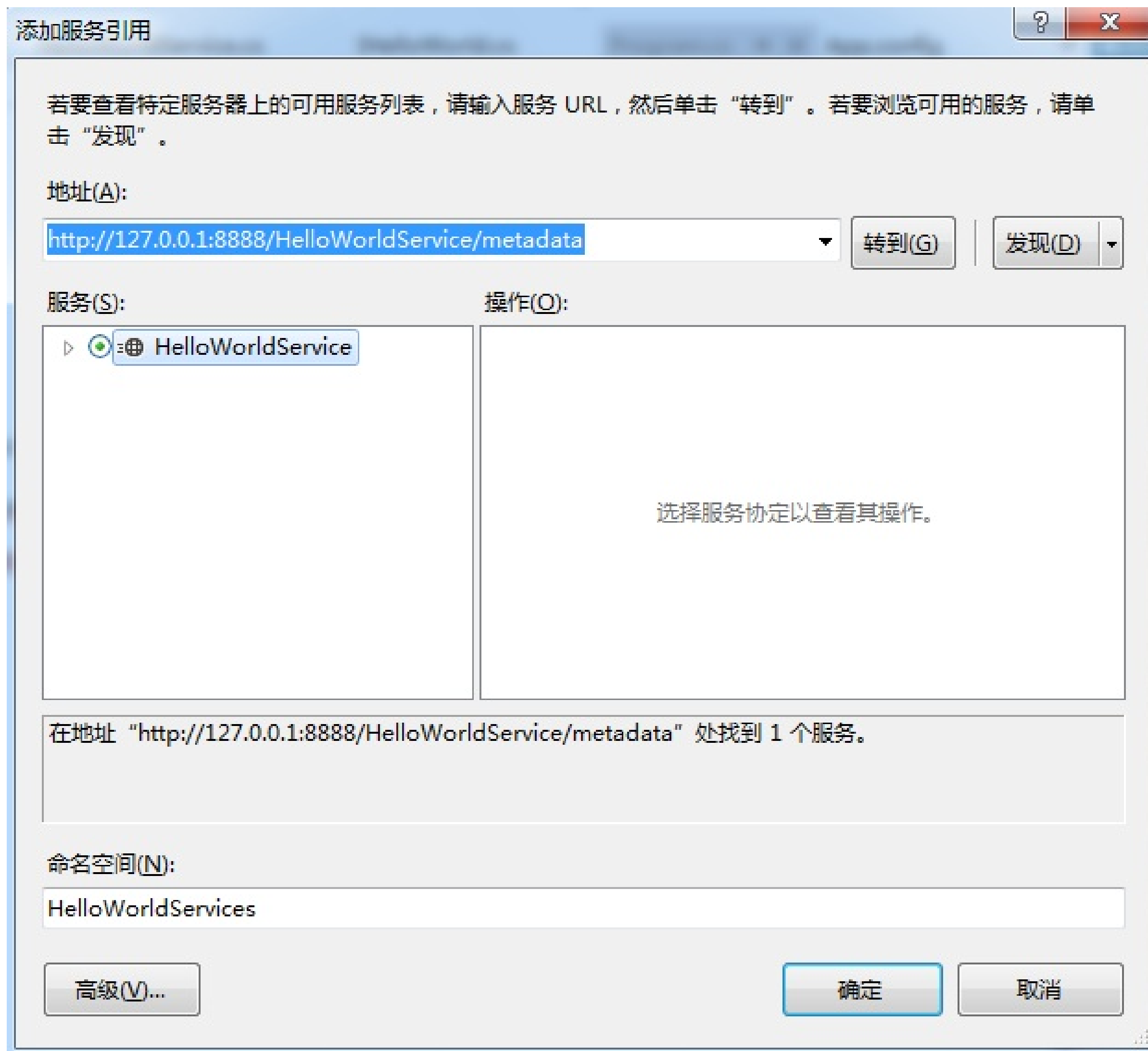
而在进行真正的WCF应用开发时，都不会直接通过硬编码的方式进行终结点的添加和服务行为的定义，而是通过配置文件的方式来进行的。你可以以下面配置文件的方式来代替上面的硬编码方式。

```
<configuration>
    <system.serviceModel>
        <behaviors>
          <serviceBehaviors>
            <behavior name="metadataBehavior">
              <serviceMetadata httpGetEnabled="true" httpGetUrl="ht
            </behavior>
          </serviceBehaviors>
        </behaviors>
      <services>
        <service behaviorConfiguration="metadataBehavior" name ="Se
          <endpoint address="http://127.0.0.1:8888/HelloWorldServic
                    binding="wsHttpBinding"
                    contract="Contract.IHelloWorld"/>
        </service>
      </services>
    </system.serviceModel>
</configuration>
```

步骤三：创建客户端

服务成功寄宿之后，服务端便开始了服务调用请求的监听工作。另外，服务寄宿将服务描述通过元数据的方式发布出来，相应的客户端就可以获取这些元数据来创建客户端程序来对服务进行调用。在Visual Studio下，当我们添加服务引用时，VS在内部会帮我们实现元数据的获取，并通过代码生成工具（SvcUtil.exe）将这些元数据自动生成用于服务调用的服务代理相关的代码和相应的配置。

在成功运行服务寄宿程序后，右键客户端项目，在弹出的菜单中选择"添加服务引用"，然后在弹出的添加服务引用窗口中输入服务元数据的地址：http://127.0.0.1:8888/HelloWorldService/metadata，并指定一个命名空间，点击确定按钮（具体效果如下图所示），VS将为你生成用于服务调用的代理类代码和配置信息。



添加成功之后，我们可以通过创建服务代理类对象来对服务相应方法进行调用操作，客户端进行服务调用的具体实现代码如下所示：

```
1 using Client.HelloWorldServices;
2 using System;
3
4 namespace Client
5 {
6     class Program
7     {
8         static void Main(string[] args)
9         {
10            // HelloworldServiceClient就是VS为我们创建的服务代理类
11            using (HellworldServiceClient proxy = new HellworldS
12            {
13                // 通过代理类来调用进行服务方法的访问
14                Console.WriteLine("服务返回的结果是: {0}", proxy.G
15            }
16
17            Console.Read();
18        }
19    }
20 }
```

运行客户端程序后，你将获得如下图所示的运行结果。



上面演示了通过自我寄宿的方式来寄宿服务，接下来我们来演示如何将WCF服务寄宿到IIS中。因为WCF服务和服务契约在上面方式中已实现，所以IIS寄宿方式的包含两个步骤：创建IIS宿主服务和创建客户端调用程序。下面分别介绍下这两个步骤。

步骤一：创建IIS宿主服务

在开始的解决方案中，创建一个Asp.net空Web应用程序，然后添加一个WCF服务文件。这里WCF服务文件与Web 服务中的.asmx文件类似。基于IIS的服务寄宿要求相应的WCF服务具有相应的.svc文件，.svc文件部署于IIS站点中，对WCF服务的调用体现在对.svc文件的访问上。

WCF服务文件的内容很简单，仅仅包含一个ServiceHost指令，该指令具有一个必须的Service属性和一些可选的属性。详细信息见MSDN：@ServiceHost.aspx)。所以对应的.svc文件内容如下所示：

具体Web.Config的配置内容如下所示：

```
<configuration>
  <system.web>
    <compilation debug="true"/>
  </system.web>
  <system.serviceModel>
    <behaviors>
      <serviceBehaviors>
        <behavior name="metadataBehavior">
          <serviceMetadata httpGetEnabled="true"/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
    <services>
      <service behaviorConfiguration="metadataBehavior" name ="Serv
        <endpoint binding="wsHttpBinding"
                  contract="Contract.IHelloWorld"/>
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

从上面配置内容可以看出，这基本上和自我寄宿方式的配置文件一致，唯一不同的是在添加终结点中无需指定地址，因为.svc所在的地址就是服务的地址。

步骤二：创建客户端程序

此时，客户端仅仅需要修改终结点地址来对寄宿于IIS下的HellworldService进行访问，该地址为：http://localhost:15826/HelloWorldService.svc。此时可以http://localhost:15826/HelloWorldService.svc?wsdl得到相应的元数据。具体客户端代码的实现如下所示：

```
 1 using Client2.HelloWorldService;
 2 using System;
 3
 4 namespace Client2
 5 {
 6     class Program
 7     {
 8         static void Main(string[] args)
 9         {
10             using (HellworldServiceClient proxy = new HellworldS
11             {
12                 Console.WriteLine("服务返回的结果是: {0}", proxy.G
13             }
14
15             Console.Read();
16         }
17     }
18 }
```

具体的配置文件内容如下所示：

```
<configuration>
    <startup>
        <supportedRuntime version="v4.0" sku=".NETFramework,Versior
    </startup>
    <system.serviceModel>
        <bindings>
            <wsHttpBinding>
                <binding name="WSHttpBinding_HellworldService" />
            </wsHttpBinding>
        </bindings>
        <client>
            <endpoint address="http://localhost:15826/HelloWorldSe
                binding="wsHttpBinding" bindingConfiguration="WSHtt
                contract="HelloWorldService.HellworldService" name=
            </endpoint>
        </client>
    </system.serviceModel>
</configuration>
```

运行客户端2，得到的运行结果与自我寄宿方式得到的结果一样。这里就不贴出结果图了。

四、总结

到这里，本篇文章的分享就结束。本文首先通过介绍WCF相关的基础概念，其中最重要的莫过于终结点和组成它的三个元素，之后分别介绍自我寄宿和IIS寄宿方式来创建WCF应用程序，在平常开发过程中，用到最多是通过IIS寄宿方式来对服务进行寄宿。在一篇文章中将分享关于WCF服务契约的内容。

本文所有源代码下载地址：FirstWCFApp.zip

跟我一起学WCF(5)——深入解析服务契约[上篇]

一、引言

在上一篇博文中，我们创建了一个简单WCF应用程序，在其中介绍到WCF最重要的概念又是终结点，而终结点又是由ABC组成的。对于Address地址也就是告诉客户端WCF服务所在的位置，而Contract又是终结点中比较重要的一个内容，在WCF中，契约包括服务契约、数据契约、消息契约和错误契约，在本篇博文将解析下数据契约的内容，关于其他三种契约将会后面的博文中陆续介绍。

二、引出问题——WCF操作重载限制

C#语言是支持操作重载的，然而在WCF实现操作重载有一定的限制。错误的操作重载实例：

```
1 [ServiceContract(Name = "HellworldService", Namespace = "http://v
2     public interface IHelloWorld
3     {
4         [OperationContract]
5         string GetHelloWorld();
6
7         [OperationContract]
8         string GetHelloWorld(string name);
9     }
```

如果你像上面一样来实现操作重载的话，在开启服务的时候，你将收到如下图所示的异常信息：

```
using (ServiceHost host = new ServiceHost(typeof(Services.HelloWorldService)))
{
    host.Opened += delegate
    {
        Console.WriteLine("服务已开启，按任意键继续.
    };

    host.Open();
    Console.ReadLine();
}
```

未处理InvalidOperationException

同一个协定中不能存在两个名称相同的操作，类型为 Contract.IHelloWorld 的方法 GetHelloWorld 和 GetHelloWorld 违反了此规则。可以通过更改方法名称或使用 OperationContractAttribute 的 Name 属性更改其中一个操作的名称。

疑难解答提示:

获取此异常的常规帮助。

搜索更多联机帮助...

异常设置:

引发此异常类型时中断

操作:

查看详细信息...

将异常详细信息复制到剪贴板

打开异常设置

然而，为什么WCF不允许定义两个相同的操作名称呢？原因很简单，因为WCF的实现是基于XML的，它是通过WSDL来进行描述，而WSDL也是一段XML。在WSDL中，WCF的一个方法对应一个操作（operation）标签。我们可以参考下面一段XML，它是从一个WCF的WSDL中截取下来的。

```
<wsdl:import namespace="http://www.Learninghard.com" location="http
<wsdl:types/>
<wsdl:binding name="BasicHttpBinding_HellworldService" type="i0:Hel
<soap:binding transport="http://schemas.xmlsoap.org/soap/http"/>
<wsdl:operation name="GetHelloWorldWithoutParam">
<soap:operation soapAction="http://www.Learninghard.com/HellworldSe
<wsdl:input>
<soap:body use="literal"/>
</wsdl:input>
<wsdl:output>
<soap:body use="literal"/>
</wsdl:output>
</wsdl:operation>
<wsdl:operation name="GetHelloWorldWithParam">
<soap:operation soapAction="http://www.Learninghard.com/HellworldSe
<wsdl:input>
<soap:body use="literal"/>
</wsdl:input>
<wsdl:output>
<soap:body use="literal"/>
</wsdl:output>
</wsdl:operation>
</wsdl:binding>
<wsdl:service name="HelloWorldService">
<wsdl:port name="BasicHttpBinding_HellworldService" binding="tns:Ba
<soap:address location="http://localhost:9999/GetHelloWorldService"
</wsdl:port>
</wsdl:service>
```

从上面的代码可以看出，每个Operation由一个operation XML Element表示，而每个Operation还应该具有一个能够唯一表示该Operation的ID，这个ID则是通过name属性来定义。Operation元素的Name属性通常使用方法名来定义，所以，如果WCF服务契约中，包含两个相同的操作方法名时，此时就违背了WSDL的规定，这也是WCF不可以使用操作重载的原因。

三、解决问题——WCF中实现操作重载

既然，找到了WCF中不能使用操作重载的原因(即必须保证Operation元素的Name属性唯一)，此时，要想实现操作重载，则有两种思路：一是两个不同的操作名，二是实现一种Mapping机制，使得服务契约中的方法名映射到一个其他的方法名，从而来保证Name属性的唯一。对于这两种解决思路，第一种显然行不通，因为，方法名不同显然就不叫操作重载了，所以，我们可以从第二种解决思路下手。值得庆幸的是，这种解决思路，微软在实现WCF的时候已经帮我们实现好了，我们可以通过OperationContractAttribute.aspx)的Name属性来为每个操作方法名定义一个别人，而生成的WSDL将使用这个别名来作为Operation元素的Name属性，所以我们只需要为两个相同方法名定义两个不同的别名就可以解决操作重载的问题了。既然有了思路，下面就看看具体的实现代码吧。具体服务契约的实现方法如下所示：

```
 1 namespace Contract
 2 {
 3     [ServiceContract(Name = "HellworldService", Namespace = "htt
 4     public interface IHelloWorld
 5     {
 6         [OperationContract(Name = "GetHelloWorldWithoutParam")]
 7         string GetHelloWorld();
 8 
 9         [OperationContract(Name = "GetHelloWorldWithParam")]
10         string GetHelloWorld(string name);
11     }
12 }
```

经过上面的步骤也就解决了在WCF中实现操作重载的问题。接下来让我们来完成一个完整的操作重载的例子。

定义契约完成之后，那就接着来实现下服务契约，具体的实现服务契约代码如下所示：

```
namespace Services
{
    public class HelloWorldService : IHelloWorld
    {
        public string GetHelloWorld()
        {
            return "Hello World";
        }

        public string GetHelloWorld(string name)
        {
            return "Hello " + name;
        }
    }
}
```

接着，来继续为这个WCF服务提供一个宿主环境，这里先以控制台应用程序来实现宿主应用程序，具体的实现代码和配置代码如下所示：

```
namespace WCFServiceHostByConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            using (ServiceHost host = new ServiceHost(typeof(Servic
            {
                host.Opened += delegate
                {
                    Console.WriteLine("服务已开启，按任意键继续....");
                };

                host.Open();
                Console.ReadLine();
            }
        }
    }
}
```

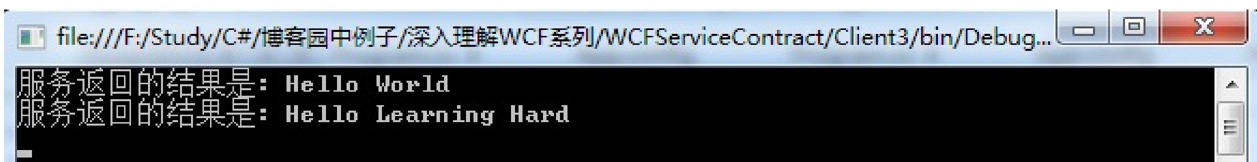
对应的服务端配置文件如下所示：

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <behaviors>
      <serviceBehaviors>
        <behavior name="HelloWorldSerBehavior">
          <serviceMetadata httpGetEnabled="True" httpGetUrl="http:/
        </behavior>
      </serviceBehaviors>
    </behaviors>
    <services>
      <service name ="Services.HelloWorldService" behaviorConfigura
        <endpoint address="http://localhost:9999/GetHelloWorldServi
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

接着，我们来创建一个客户端通过代理对象来调用WCF服务方法。首先以管理员运行WCFServiceHostByConsoleApp.exe文件来开启服务，WCF服务开启成功后，在对应的客户端右键添加服务引用，在打开的添加服务引用窗口中输入WCF服务地址：http://localhost:9999/GetHelloWorldService，点确定按钮来添加服务引用，添加成功后，VS中集成的代码生成工具会帮我们生成对应的代理类。接下来，我们可以通过创建一个代理对象来对WCF进行访问。具体客户端的实现代码如下所示：

```
 1 namespace Client3
 2 {
 3     class Program
 4     {
 5         static void Main(string[] args)
 6         {
 7             using (HellworldServiceClient helloWorldProxy = new
 8             {
 9                 Console.WriteLine("服务返回的结果是: {0}", helloWo
10                 Console.WriteLine("服务返回的结果是: {0}", helloWo
11             }
12 
13             Console.ReadLine();
14         }
15     }
16 }
```

这样，你运行客户端程序时（注意不要关闭WCF服务宿主程序），你将看到如下图所示的运行结果。



在上面客户端的实现代码中，从客户端的角度来看，我们并不知道我们是对重载方法进行调用，因为我们调用的明明是两个不同的方法名，这显然还不是我们最终想要达到的效果，此时，有两种方式来达到客户端通过相同方法名来调用。

- 第一种方式就是手动修改生成的服务代理类和服务契约代码，使其支持操作重载，修改后的服务代理和服务契约代码如下所示：

```
namespace Client3.ServiceReference {

    [System.CodeDom.Compiler.GeneratedCodeAttribute("System.Service
    [System.ServiceModel.ServiceContractAttribute(Namespace="http:/
    public interface HellworldService {

        // 把自动生成的方法名GetHelloWorldWithoutParam修改成GetHelloWor
        [System.ServiceModel.OperationContractAttribute(Name = "Get
        string GetHelloWorld();

        // // 把自动生成的方法名GetHelloWorldWithoutParamAsync修改成Get
        [System.ServiceModel.OperationContractAttribute(Name = "Get
        System.Threading.Tasks.Task<string> GetHelloWorldAsync();

        [System.ServiceModel.OperationContractAttribute(Name = "Get
        string GetHelloWorld(string name);

        [System.ServiceModel.OperationContractAttribute(Name = "Get
        System.Threading.Tasks.Task<string> GetHelloWorldAsync(stri
    }

    [System.CodeDom.Compiler.GeneratedCodeAttribute("System.Service
    public interface HellworldServiceChannel : Client3.ServiceRefer
    }

    [System.Diagnostics.DebuggerStepThroughAttribute()]
    [System.CodeDom.Compiler.GeneratedCodeAttribute("System.Service
    public partial class HellworldServiceClient : System.ServiceMod

        public HellworldServiceClient() {
        }

        public HellworldServiceClient(string endpointConfigurationN
                base(endpointConfigurationName) {
        }

        public HellworldServiceClient(string endpointConfigurationN
                base(endpointConfigurationName, remoteAddress) {
        }
```

```
        public HellworldServiceClient(string endpointConfigurationN
                base(endpointConfigurationName, remoteAddress) {
        }

        public HellworldServiceClient(System.ServiceModel.Channels.
                base(binding, remoteAddress) {
        }

        public string GetHelloWorld() {
            return base.Channel.GetHelloWorld();
        }

        public System.Threading.Tasks.Task<string> GetHelloWorldAsy
            return base.Channel.GetHelloWorldAsync();
        }

        public string GetHelloWorld(string name) {
            return base.Channel.GetHelloWorld(name);
        }

        public System.Threading.Tasks.Task<string> GetHelloWorldAsy
            return base.Channel.GetHelloWorldAsync(name);
        }
    }
}
```

此时，客户端的实现代码如下所示：

```
class Program
    {
        static void Main(string[] args)
        {
            using (HellworldServiceClient helloWorldProxy = new Hel
            {
                Console.WriteLine("服务返回的结果是: {0}", helloWorld
                Console.WriteLine("服务返回的结果是: {0}", helloWorld
            }

            Console.ReadLine();
        }
    }
```

此时，客户端运行后的运行结果与上面的运行结果一样，这里就不贴图了。

- 第二种方式就是自己实现客户端代理类，而不是由VS代码生成工具。具体重新的proxy Class的实现代码代码如下所示：

```
 1 using Contract;
 2 using System.ServiceModel;
 3 namespace Client2
 4 {
 5     class HellworldServiceClient : ClientBase<IHelloWorld>, IHel
 6     {
 7         #region IHelloWorld Members
 8         public string GetHelloWorld()
 9         {
10             return this.Channel.GetHelloWorld();
11         }
12
13         public string GetHelloWorld(string name)
14         {
15             return this.Channel.GetHelloWorld(name);
16         }
17         #endregion
18     }
19 }
```

此时客户端的实现代码和配置文件如下所示：

```
namespace Client2
{
    class Program
    {
        static void Main(string[] args)
        {
            using (var proxy = new HellworldServiceClient())
            {
                // 通过自定义代理类来调用进行服务方法的访问
                Console.WriteLine("服务返回的结果是: {0}", proxy.GetH
                Console.WriteLine("服务返回的结果是: {0}", proxy.GetH
            }

            Console.Read();
        }
    }
}
```

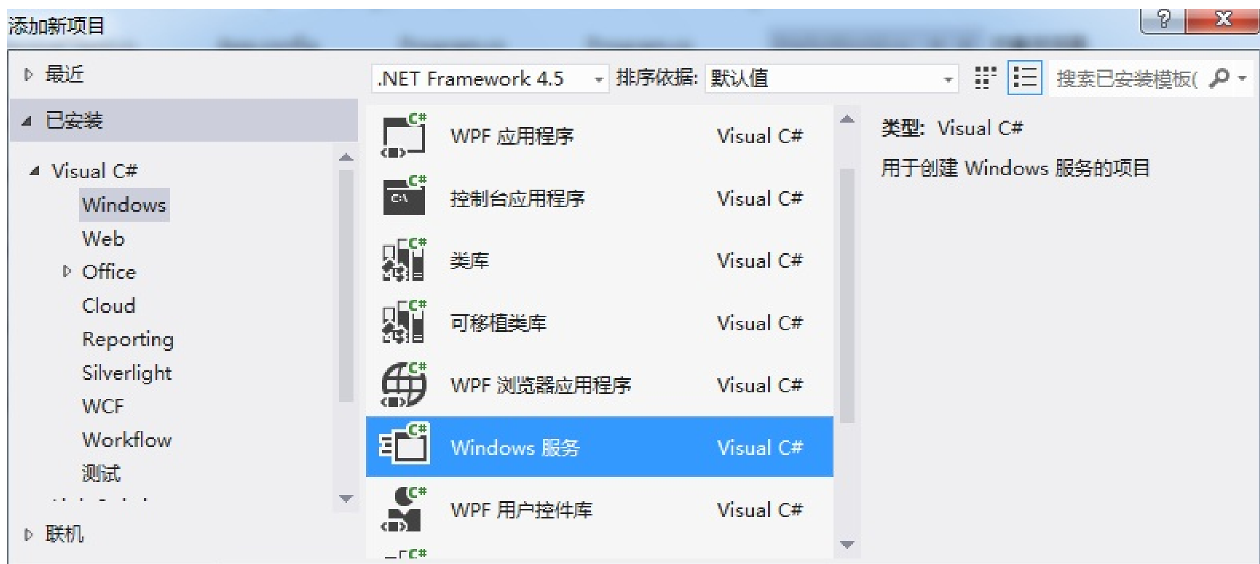
对应的配置文件如下所示：

```
<configuration>
  <system.serviceModel>
    <client>
      <endpoint address="http://localhost:9999/GetHelloWorldService
                binding ="basicHttpBinding"
                contract ="Contract.IHelloWorld"/>
    </client>
  </system.serviceModel>
</configuration>
```

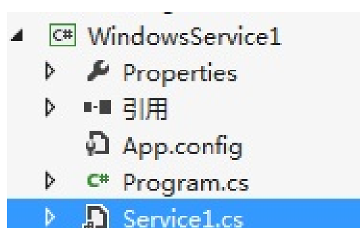
四、利用Windows Service来寄宿WCF服务

在上一篇博文中，我们介绍了把WCF寄宿在控制台应用程序和IIS中，而WCF服务可以寄宿在任何应用程序中，如WPF、WinForms和Windows Services。这里再实现下如何在Windows Services中寄宿WCF服务。下面一步步来实现该目的。

- 第一步：创建Windows 服务项目，具体添加步骤为右键解决方案->添加->新建项目，在已安装模板中选择Windows 服务模板，具体如下图示所示：



- 第二步：添加Windows服务之后，你将看到如下图所示的目录结构。



然后修改对应的Service1.cs文件，使其实现如下代码所示：

```
1 // 修改类名
2 public partial class WindowsService : ServiceBase
3     {
4         public WindowsService()
5         {
6             InitializeComponent();
7         }
8 
9         public ServiceHost serviceHost = null;
10 
11         // 启动Windows服务
12         protected override void OnStart(string[] args)
13         {
14             if (serviceHost != null)
15             {
16                 serviceHost.Close();
17             }
18 
19             serviceHost = new ServiceHost(typeof(Services.HelloW
20             serviceHost.Open();
21         }
22 
23         // 停止Windows服务
24         protected override void OnStop()
25         {
26             if (serviceHost != null)
27             {
28                 serviceHost.Close();
29                 serviceHost = null;
30             }
31         }
32     }
```

对应的配置文件代码如下所示：

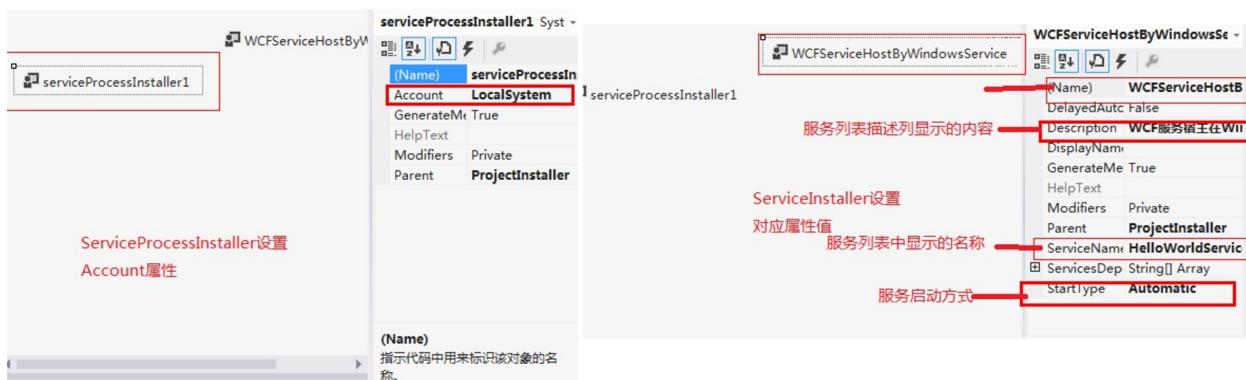
```
<system.serviceModel>
      <behaviors>
        <serviceBehaviors>
          <behavior name="WindowsServiceBehavior">
            <serviceMetadata httpGetEnabled="true" />
            <serviceDebug includeExceptionDetailInFaults="tru
          </behavior>
        </serviceBehaviors>
      </behaviors>
      <services>
        <service name="Services.HelloWorldService" behaviorConf
          <endpoint address=""
            binding="wsHttpBinding" bindingConfiguration=""
            contract="Contract.IHelloWorld" />
          <host>
            <baseAddresses>
              <add baseAddress="http://localhost:8888/WCFServic
            </baseAddresses>
          </host>
        </service>
      </services>
   </system.serviceModel>
```

- 第三步：在WindowsService的设计界面，右键选择添加安装程序，具体操作如下图所示。

若要在类中添加组件，请从工具箱中拖出它们，然后使用“属性”窗口来设置它们的属性。若要为类创建方法和事件，请单击此处切换到代码视图。



添加安装程序之后，会多出一个ProjectInstaller.cs文件，然后在其设计页面修改ServiceProcessInstaller.aspx)和ServiceInstaller.aspx)对象属性，具体设置的值如下图所示：



经过上面的步骤，程序的代码就都已经全部实现了，接下来要做的是安装Windows服务和启动Windows服务。

首先是安装Windows服务：以管理员身份运行VS2012开发命令提示，进入项目的对应的exe所在的文件夹，这里的指的是WindowsServiceHost.exe所在的文件夹，然后运行 “**installutil WindowsServiceHost.exe**”命令，命令运行成功后，你将看到如下所示的运行结果：

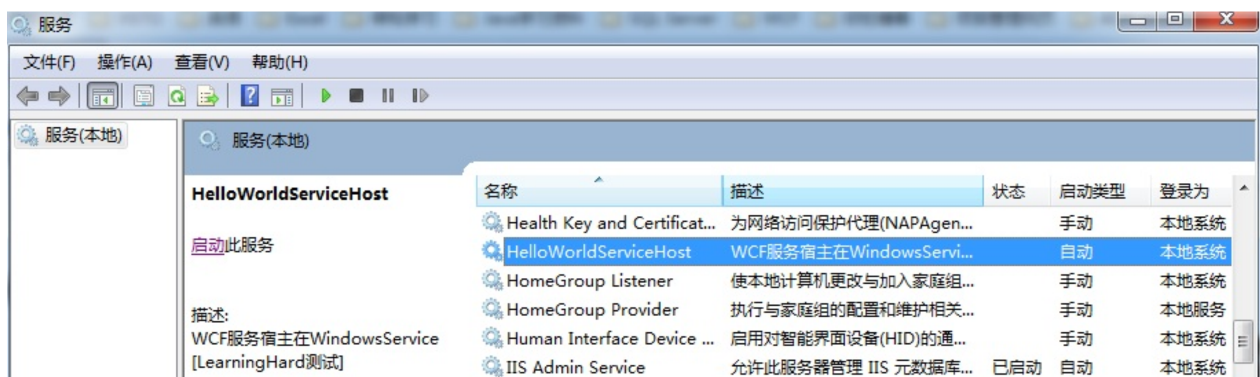
管理员: VS2012 开发人员命令提示

```
F:\Study\C#\博客园中例子\深入理解WCF系列\WCFServiceContract\WindowsServiceHost\bin\Debug>installutil WindowsServiceHost.exe
Microsoft (R) .NET Framework 安装实用工具版本 4.0.30319.18408
版权所有 (C) Microsoft Corporation。保留所有权利。


正在运行事务处理安装。

正在开始安装的“安装”阶段。
查看日志文件的内容以获得 F:\Study\C#\博客园中例子\深入理解WCF系列\WCFServiceContract\WindowsServiceHost\bin\Debug\WindowsServiceHost.exe 程序集的进度。
该文件位于 F:\Study\C#\博客园中例子\深入理解WCF系列\WCFServiceContract\WindowsServiceHost\bin\Debug\WindowsServiceHost.InstallLog。
正在安装程序集“F:\Study\C#\博客园中例子\深入理解WCF系列\WCFServiceContract\WindowsServiceHost\bin\Debug\WindowsServiceHost.exe”。
受影响的参数是:
   logtoconsole =
   logfile = F:\Study\C#\博客园中例子\深入理解WCF系列\WCFServiceContract\WindowsServiceHost\bin\Debug\WindowsServiceHost.InstallLog
   assemblypath = F:\Study\C#\博客园中例子\深入理解WCF系列\WCFServiceContract\WindowsServiceHost\bin\Debug\WindowsServiceHost.exe
正在安装服务 HelloWorldServiceHost...
已成功安装服务 HelloWorldServiceHost。
正在日志 Application 中创建 EventLog 源 HelloWorldServiceHost...
```

安装成功之后，你可以运行 “net start HelloWorldServiceHost” 命令来启动服务。因为开始设置服务的名称是HelloWorldServiceHost。你也可以通过Services中来手动启动服务，启动成功之后，你将在服务窗口看到启动的服务。具体效果如下图所示。



服务启动后，在客户端中同样是添加服务引用的方式来添加服务引用，在添加服务引用窗口输入地址：http://localhost:8888/WCFServiceHostByWindowsService。点击确定按钮。添加服务引用成功后，对应的客户端调用代码如下所示：

```
 1 namespace Client
 2 {
 3     class Program
 4     {
 5         static void Main(string[] args)
 6         {
 7             using (var proxy = new HellworldServiceClient())
 8             {
 9                 // 通过代理类来调用进行服务方法的访问
10                 Console.WriteLine("服务返回的结果是: {0}", proxy.G
11                 Console.WriteLine("服务返回的结果是: {0}", proxy.G
12             }
13 
14             Console.Read();
15         }
16 
17     }
18 }
```

此时的运行结果和前面客户端返回的运行结果是一样的。

五、总结

到这里，本文的内容就介绍结束了，本文主要解决了在WCF中如何实现操作重载的问题，实现思路可以概括为利用OperationContractAttribute类的Name属性来实现操作重载，而客户端的实现思路可以概括为重新代理类，利用信道Channel类带对对应的服务方法进行调用，最后，实现了把WCF服务寄宿在Windows Services中，这样WCF服务可以作为服务在机器上设置开机启动或其他方式启动了。在下一篇博文中将分享WCF服务契约的继承实现。

本人所有源代码下载：WCFServiceContract.zip

跟我一起学WCF(6)——深入解析服务契约[下篇]

一、引言

在上一篇博文中，我们分析了如何在WCF中实现操作重载，其主要实现要点是服务端通过ServiceContract的Name属性来为操作定义一个别名来使操作名不一样，而在客户端是通过重写客户端代理类的方式来实现的。在这篇博文中将分享契约继承的实现。

二、WCF服务契约继承实现的限制

首先，介绍下WCF中传统实现契约继承的一个方式，下面通过一个简单的WCF应用来看看不做任何修改的情况下是如何实现契约继承的。我们还是按照之前的步骤来实现下这个WCF应用程序。

- 步骤一：实现**WCF**服务

在这里，我们定义了两个服务契约，它们之间是继承的关系的，具体的实现代码如下所示：

```
 1  // 服务契约
 2     [ServiceContract]
 3     public interface ISimpleInstrumentation
 4     {
 5         [OperationContract]
 6         string WriteEventLog();
 7     }
 8
 9 // 服务契约，继承于ISimpleInstrumentation这个服务契约
10     [ServiceContract]
11     public interface ICompleteInstrumentation :ISimpleInstrument
12     {
13         [OperationContract]
14         string IncreatePerformanceCounter();
15     }
```

上面定义了两个接口来作为服务契约，其中ICompleteInstrumentation继承ISimpleInstrumentation。这里需要注意的是：虽然ICompleteInstrumentation继承于ISimpleteInstrumentation，但是运用在ISimpleInstrumentation中的ServiceContractAttribute却不能被ICompleteInstrumentation继承，这是因为在它之上的AttributeUsage的Inherited属性设置为false，代表ServiceContractAttribute.aspx)不能被派生接口继承。ServiceContractAttribute的具体定义如下所示：

接下来实现对应的服务，具体的实现代码如下所示：

```
// 实现ISimpleInstrumentation契约
   public class SimpleInstrumentationService : ISimpleInstrumentat
   {
       #region ISimpleInstrumentation members
       public string WriteEventLog()
       {
          return "Simple Instrumentation Service is Called";
       }
       #endregion
   }

// 实现ICompleteInstrumentation契约
   public class CompleteInstrumentationService: SimpleInstrumentat
   {
       public string IncreatePerformanceCounter()
       {
           return "Increate Performance Counter is called";
       }
   }
```

上面中，为了代码的重用，CompleteInstrumentationService继承自SimpleInstrumentationService，这样就不需要重新定义WriteEventLog方法了。

- 步骤二：实现服务宿主

定义完成服务之后，现在就来看看服务宿主的实现，这里服务宿主是一个控制台应用程序，具体实现代码与前面几章介绍的代码差不多，具体的实现代码如下所示：

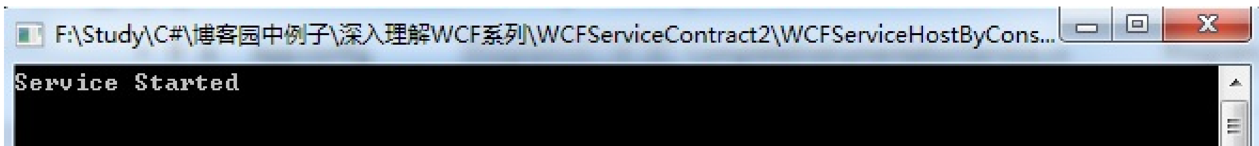
```
1 // 服务宿主的实现，把WCF服务宿主在控制台程序中
2     class Program
3     {
4         static void Main(string[] args)
5         {
6             using (ServiceHost host = new ServiceHost(typeof(WCF
7             {
8                 host.Opened += delegate
9                 {
10                    Console.WriteLine("Service Started");
11                };
12
13                // 打开通信通道
14                host.Open();
15                Console.Read();
16            }
17
18        }
19    }
```

宿主程序对应的配置文件信息如下所示：

```
<configuration>
    <system.serviceModel>
        <behaviors>
            <serviceBehaviors>
                <behavior name="metadataBehavior">
                    <serviceMetadata httpGetEnabled="true"/>
                </behavior>
            </serviceBehaviors>
        </behaviors>
        <services>
          <!--service标签的Name属性是必须，而且必须指定为服务类型，指定格式
          <!--更多信息可以参考MSDN：http://msdn.microsoft.com/zh-cn/li
            <service name="WCFService.CompleteInstrumentationServic
                <endpoint address="mex" binding="mexHttpBinding" co
                <host>
                    <baseAddresses>
                        <add baseAddress="http://localhost:9003/ins
                    </baseAddresses>
                </host>
            </service>
        </services>
    </system.serviceModel>
</configuration>
```

- 步骤三：实现客户端

最后，就是实现我们的客户端来对服务进行访问了，这里首先以管理员权限运行宿主应用程序，即以管理员权限运行WCFServiceHostByConsoleApp.exe可执行文件。运行成功之后，你将在控制台中看到服务启动成功的消息，具体运行结果如下图所示：



然后在客户端通过添加服务引用的方式来添加服务引用，此时必须记住，一定要先运行宿主服务，这样才能在添加服务引用窗口中输入地址：http://localhost:9003/instrumentationService/ 才能获得服务的元数据信息。添加成功后，svcutil.exe工具除了会为我们生成对应的客户端代理类之前，还会自动添加配置文件信息，而且还会为我们添加System.ServiceModel.dll的引用。下面就是工具为我们生成的代码：

```
[System.CodeDom.Compiler.GeneratedCodeAttribute("System.ServiceMode
    [System.ServiceModel.ServiceContractAttribute(ConfigurationName
    public interface **ICompleteInstrumentation** {

        [System.ServiceModel.OperationContractAttribute(Action="htt
        string WriteEventLog();

        [System.ServiceModel.OperationContractAttribute(Action="htt
        System.Threading.Tasks.Task<string> WriteEventLogAsync();

        [System.ServiceModel.OperationContractAttribute(Action="htt
        string IncreatePerformanceCounter();

        [System.ServiceModel.OperationContractAttribute(Action="htt
        System.Threading.Tasks.Task<string> IncreatePerformanceCour
    }

    [System.CodeDom.Compiler.GeneratedCodeAttribute("System.Service
    public interface ICompleteInstrumentationChannel : ClientConsol
    }

    [System.Diagnostics.DebuggerStepThroughAttribute()]
    [System.CodeDom.Compiler.GeneratedCodeAttribute("System.Service
    public partial class **CompleteInstrumentationClient** : Syster

        public CompleteInstrumentationClient() {
        }

        public CompleteInstrumentationClient(string endpointConfigu
                base(endpointConfigurationName) {
        }

        public CompleteInstrumentationClient(string endpointConfigu
                base(endpointConfigurationName, remoteAddress) {
        }
```

```
        public CompleteInstrumentationClient(string endpointConfigu
                base(endpointConfigurationName, remoteAddress) {
        }

        public CompleteInstrumentationClient(System.ServiceModel.Ch
                base(binding, remoteAddress) {
        }

        public string WriteEventLog() {
            return base.Channel.WriteEventLog();
        }

        public System.Threading.Tasks.Task<string> WriteEventLogAsy
            return base.Channel.WriteEventLogAsync();
        }

        public string IncreatePerformanceCounter() {
            return base.Channel.IncreatePerformanceCounter();
        }

        public System.Threading.Tasks.Task<string> IncreatePerforma
            return base.Channel.IncreatePerformanceCounterAsync();
        }
    }
```

在服务端，我们定义了具有继承层次结构的服务契约，并为ICompleteInstrumentation契约公开了一个EndPoint。但是在客户端，我们通过添加服务引用的方式生成的服务契约却没有了继承的关系，在上面代码标红的地方可以看出，此时客户端代理类中只定义了一个服务契约，在该服务契约定义了所有的Operation。此时客户端的实现代码如下所示：

```
 1 class Program
 2     {
 3         static void Main(string[] args)
 4         {
 5             Console.WriteLine("---Use Genergate Client by VS Too
 6             using (CompleteInstrumentationClient proxy = new Com
 7             {
 8                 Console.WriteLine(proxy.WriteEventLog());
 9                 Console.WriteLine(proxy.IncreatePerformanceCount
10             }
11 
12             Console.Read();
13         }
14     }
```

从上面代码可以看出。虽然现在我们可以通过调用CompleteInstrumentationClient代理类来完成服务的调用，但是我们希望的是，客户端代理类也具有继承关系的契约结构。

三、实现客户端的契约层级

既然，自动生成的代码不能完成我们的需要，此时我们可以通过自定义的方式来定义自己的代理类。

- 第一步就是定义客户端的Service Contract。具体的自定义代码如下所示：

```
 1 namespace ClientConsoleApp
 2 {
 3     // 自定义服务契约，使其保持与服务端一样的继承结果
 4     [ServiceContract]
 5     public interface ISimpleInstrumentation
 6     {
 7         [OperationContract]
 8         string WriteEventLog();
 9     }
10 
11     [ServiceContract]
12     public interface ICompleteInstrumentation : ISimpleInstrumen
13     {
14         [OperationContract]
15         string IncreatePerformanceCounter();
16     }
17 }
```

- 第二步：自定义两个代理类，具体的实现代码如下所示：

```
// 自定义代理类
   public class SimpleInstrumentationClient : ClientBase<IComplete
   {
       #region ISimpleInstrumentation Members
       public string WriteEventLog()
       {
           return this.Channel.WriteEventLog();
       }
       #endregion
   }

public class CompleteInstrumentationClient:SimpleInstrumentationCl
   {
       public string IncreatePerformanceCounter()
       {
           return this.Channel.IncreatePerformanceCounter();
       }
   }
```

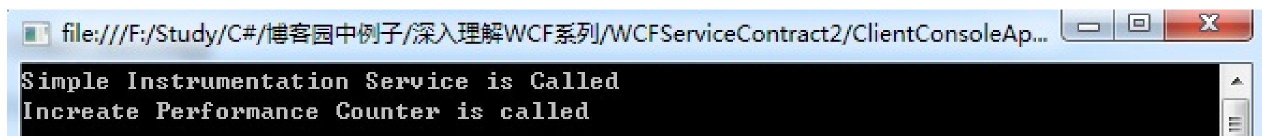
对应的配置文件修改为如下所示：

```
<configuration>
    <system.serviceModel>
        <bindings>
            <wsHttpBinding>
                <binding name="MetadataExchangeHttpBinding_IComplet
                    <security mode="None" />
                </binding>
            </wsHttpBinding>
        </bindings>
        <client>
            <endpoint address="http://localhost:9003/instrumentatio
                binding="wsHttpBinding" bindingConfiguration="Metad
                contract="ClientConsoleApp.ICompleteInstrumentation
        </client>
    </system.serviceModel>
</configuration>
```

- 第三步：实现客户端来进行服务调用，此时可以通过两个自定义的代理类来分别对两个服务契约对应的操作进行调用，具体的实现代码如下所示：

```
1 class Program
2     {
3         static void Main(string[] args)
4         {
5             using (SimpleInstrumentationClient proxy1 = new Simp
6             {
7                 Console.WriteLine(proxy1.WriteEventLog());
8             }
9             using (CompleteInstrumentationClient proxy2 = new Co
10            {
11                Console.WriteLine(proxy2.IncreatePerformanceCoun
12            }
13
14            Console.Read();
15        }
16    }
```

这样，通过重写代理类的方式，客户端可以完全以面向对象的方式调用了服务契约的方法，具体的运行效果如下图所示：



另外，如果你不想定义两个代理类的话，你也可以通过下面的方式来对服务契约进行调用，具体的实现步骤为：

- 第一步：同样是实现具有继承关系的服务契约，具体的实现代码与前面一样。

```
// 自定义服务契约，使其保持与服务端一样的继承结果
   [ServiceContract]
   public interface ISimpleInstrumentation
   {
       [OperationContract]
       string WriteEventLog();
   }

   [ServiceContract]
   public interface ICompleteInstrumentation : ISimpleInstrumentat
   {
       [OperationContract]
       string IncreatePerformanceCounter();
   }
```

- 第二步：配置文件修改。把客户端配置文件修改为如下所示：

```
<configuration>
    <system.serviceModel>
        <client>
            <endpoint address="http://localhost:9003/instrumentatic
                binding="mexHttpBinding" contract="ClientConsoleApp
                name="ISimpleInstrumentation" />
            <endpoint address="http://localhost:9003/instrumentatic
                binding="mexHttpBinding" contract="ClientConsoleApp
                name="ICompleteInstrumentation" />
        </client>
    </system.serviceModel>
</configuration>
```

- 第三步：实现客户端代码。具体的实现代码如下所示：

```
 1 class Program
 2     {
 3         static void Main(string[] args)
 4         {
 5             using (ChannelFactory<ISimpleInstrumentation> simple
 6             {
 7                 ISimpleInstrumentation simpleProxy = simpleChann
 8                 using (simpleProxy as IDisposable)
 9                 {
10                     Console.WriteLine(simpleProxy.WriteEventLog(
11                 }
12             }
13             using (ChannelFactory<ICompleteInstrumentation> comp
14             {
15                 ICompleteInstrumentation completeProxy = complet
16                 using (completeProxy as IDisposable)
17                 {
18                     Console.WriteLine(completeProxy.IncreatePerf
19                 }
20             }
21 
22             Console.Read();
23         }
24     }
```

其实，上面的实现代码原理与定义两个客户端代理类是一样的，只是此时把代理类放在客户端调用代码中实现。通过上面代码可以看出，要进行通信，主要要创建与服务端的通信信道，即Channel，上面的是直接通过ChannelFactory<T>的CreateChannel方法来创建通信信道，而通过定义代理类的方式是通过ClientBase<T>的Channel属性来获得当前通信信道，其在ClientBase类本身的实现

也是通过ChannelFactory.CreateChannel方法来创建信道的，再把这个创建的信道赋值给Channel属性，以供外面进行获取创建的信道。所以说这两种实现方式的原理都是一样的，并且通过自动生成的代理类也是一样的原理。

四、总结

到这里，本篇文章分享的内容就结束了，本文主要通过自定义代理类的方式来对契约继承服务的调用。其实现思路与上一篇操作重载的实现思路是一样的，既然客户端自动生成的代码类不能满足需求，那就只能自定义来扩展了。到此，服务契约的分享也就告一段落了，后面的一篇博文继续分享WCF中数据契约。

本人所有源码下载：WCFServiceContract2.zip。

跟我一起学WCF(7)——WCF数据契约与序列化详解

一、引言

在前面博文介绍到，WCF的契约包括操作契约、数据契约、消息契约和错误契约，前面一篇博文已经结束了操作契约的介绍，接下来自然就是介绍数据契约了。所以本文要分享的内容就是数据契约。

二、数据契约的介绍

在WCF中，服务契约定义了可供调用的服务操作方法，而数据契约则是定义了服务端和客户端之间传送的自定义类型，在WCF项目中，必不可少地是传递数据，把客户端需要传递的数据传送到服务中，服务接收到数据再对其进行处理。然而在WCF中，传递的类型必须标记为DataContractAttribute.aspx)属性，且只有标记了DataMemberAttribute属性的属性才会被传送。下面代码是一个数据契约使用的示例：

```
1 [DataContract] // 数据契约属性声明
2     public class User
3     {
4         [DataMember(Name = "UserName")]//定义别名
5         public string Name
6         { get; set; }
7         [DataMember]
8         public string Password { get; set; }
9         [DataMember]
10        public string Email { get; set; }
11
12        // 没有[DataMember]声明将不会序列化传送
13        public string Mobile { get; set; }
14
15        public string Test { get; set; }
16    }
```

上面代码在类User上使用了DataContract属性声明，则表明User类是可被WCF序列化程序可识别，并且可被序列化的。但是不是User所有数据成员都可以被需要列，只有声明了DataMemberAttribute的属性才可以被序列化。因此，在上面代码中，不会传输Mobile和Test的任何信息。同时也可以为声明为DataMember的成员定义客户端可见的别名，如DataMember(Name= "UserName")，这样在生成客户端代码时，User类定义的就是UserName属性，而不是在服务中定义的Name属性了。

三、序列化的详细介绍

WCF的实现原理沿用了.NET Remoting的实现机制，客户端在调用用服务公开的服务方法，这个过程必然涉及到数据的传输过程，包括客户端传输相关需要处理的数据给服务或服务传输相关处理后的结果数据给客户端。在数据传输的过程中，自然就需要进行序列化的操作，通过序列化把.NET Object序列化成可保存或传输的形式，然后通过网络协议在网络上进行传递。对于序列化的实现是由序列化器（Serializer）来负责完成的，序列化的实现原理可以理解为通过反射机制分析程序集中对应的类型，然后把对应的类型映射为一个XML的结构。

序列化在.NET Framework相关专题就有所介绍，所以它并不是一个新的概念，相关内容可以参考MSDN：序列化.aspx)。然而.NET本身的序列化机制在WCF程序中并不适应，所以WCF又提出了新的序列化器。下面分别介绍下.NET 序列化机制和WCF中序列化机制。

3.1 .NET序列化机制

在.NET Framework 3.0之前，提供了3中序列化器，序列化器理解为把可序列化的类型序列化成XML的类。这三种序列化器分别是BinaryFormatter.aspx)、SoapFormatter.aspx)和XmlSerializer.aspx)类。下面分别介绍下这3种序列化器。

- BinaryFormatter类：把.NET Object序列化成二进制格式。在这个过程，对象的公共字段和私有字段以及类名称（包括类的程序集名），将转换成成字节流。
- SoapFormatter类：把.NET Object序列化成SOAP格式，SOAP是一种轻量、简单的，基于XML的协议。只序列化字段，包括公共字段和私有字段。
- XmlSerializer类：该类仅仅序列化公共字段和属性，且不保存类型的保真度。

对于这三种序列化机制，BinaryFormatter二进制序列化的优点是：性能高，但是不能跨平台。而SoapFormatter，XmlSerializer的优点是：跨平台、互操作性好，并且可读性强，但是传输性能不及BinaryFormatter。

在.NET原有的序列化机制中，BinaryFormatter和SoapFormatter除了要序列化对象的状态信息外，还会将程序集和版本信息持久化到流中，因为只有这样才能保证对象呗反序列为正确的对象类型副本，这就要求客户端必须拥有原有的.NET 程序集，不能满足跨平台的需求。所以WCF不得不定义自己的序列化机制来满足面向服务的需求。

3.2 WCF中序列化机制

在WCF中，提供了专门用来序列化和反序列操作的类，该类就是DataContractSerializer.aspx)类。一般而言，WCF会自动选择使用DataContractSerializer来对可序列话数据契约进行序列化，不需要开发者直接调用。WCF除了支持DataContractSerializer类来进行序列化外，还支持另外两种序列化器，这两种序列化器分别为：XMLSerializer（定义在System.XML.Serialization

namespace）和NetDataContractSerializer.aspx) (定义在System.XML.Serialization namespace)。XmlSerializer类不是WCF专用的类，Asp.net Web服务统一使用该类作为序列化器，但XmlSerializer类支持的类少于DataContractSerializer列支持的类型，但它允许对生成的XML进行更多的控制，并且支持更多的XML架构定义语言（XSD）标准。它不需要在可序列化类上有任何声明性的属性。

DataContractSerializer class to serialize data types.">默认情况下，WCF 使用 DataContractSerializer.aspx) 类来序列化数据类型。 此序列化程序支持下列类型：

- XmlElement and DateTime, which are treated as primitives.">基元类型（如：整数、字符串和字节数组）以及某些特殊类型（如 XmlElement.aspx) 和 DateTime.aspx)）。
- DataContractAttribute attribute).">数据协定类型（用 DataContractAttribute.aspx) 属性标记的类型）。
- SerializableAttribute attribute, which include types that implement the ISerializable interface.">用 SerializableAttribute.aspx) 属性标记的类型，包括实现 ISerializable.aspx) 接口的类型。
- IXmlSerializable interface.">实现 IXmlSerializable.aspx) 接口的类型。
- 许多常见集合类型，包括许多泛型集合类型。

DataContractSerializer类与NetDataContractSerializer类类似，它们之间主要的区别在于：在使用NetDataContractSerializer进行序列化时，不需要指定序列化的类型，如：

```
NetDataContractSerializer serializer =
    new NetDataContractSerializer();  // 不需要明确指定序列化的类型
serializer.WriteObject(writer, p);

// 而使用DataContractSerializer需要明确指定序列化的类型
DataContractSerializer serializer =
            new DataContractSerializer(**typeof(Order)**); // 需要明
        serializer.WriteObject(writer, p);
```

四、WCF数据契约使用例子

介绍了那么多关于数据契约和序列化内容的介绍，下面看看数据契约具体使用的例子。

要使用数据契约，自然第一步是定义数据契约，具体数据契约的定义如下所示：

```
namespace BusinessEntity
{
    [DataContract]// 数据契约属性声明
    public class User
    {
        [DataMember(Name = "UserName")]//定义别名
        public string Name
        { get; set; }
        [DataMember]
        public string Password { get; set; }
        [DataMember]
        public string Email { get; set; }

        // 没有[DataMember]声明将不会序列化传送
        public string Mobile { get; set; }

        public string Test { get; set; }
    }
}
```

第二步：定义完数据契约后，接下来就要定义我们的服务契约和服务契约的实现了。具体的实现代码如下所示：

```
// 服务契约
    [ServiceContract]
    //[ServiceKnownType(typeof(Order))] // 这是为了演示WCF已知类型
    public interface IUserValidationService
    {
        [OperationContract]
        bool AddNewUser(User user);

        [OperationContract]
        User GetUserByName(string name);

        // 为了演示已知类型的操作方法
        //[OperationContract]
        //[ServiceKnownType(typeof(Order))]
        //bool AddOrder(OrderBase order);
    }
// 服务契约的实现
    public class UserValidationService : IUserValidationService
    {
        public bool AddNewUser(User user)
        {
            return true;
        }

        public User GetUserByName(string name)
        {
            User user = new User { Name = name, Password = "123", E
            return user;
        }

        // 演示已知类型的操作方法
        //public bool AddOrder(OrderBase order)
        //{
        //    return true;
        //}
    }
```

对应的配置文件代码为：

```
 <system.serviceModel>

    <behaviors>
      <serviceBehaviors>
        <behavior name="UserServiceBehavior">
          <!-- To avoid disclosing metadata information, set the va
          <serviceMetadata httpGetEnabled="true" httpsGetEnabled="t
          <!-- To receive exception details in faults for debugging
          <serviceDebug includeExceptionDetailInFaults="false"/>

        </behavior>
      </serviceBehaviors>
    </behaviors>
    <protocolMapping>
        <add binding="basicHttpsBinding" scheme="https" />
    </protocolMapping>
    <serviceHostingEnvironment aspNetCompatibilityEnabled="true" mu

      <services>
          <service name="WCFServiceAndHost.UserValidationService" b
              <endpoint address="" binding="wsHttpBinding" contract
          </service>
      </services>
  </system.serviceModel>
```

第三步：定义完服务之后，接下来就需要实现我们的客户端来访问服务方法了。首先，通过添加服务引用的方式来生成服务客户端代理类，生成的代理类中，User的定义如下代码所示：

```
[System.Diagnostics.DebuggerStepThroughAttribute()]
    [System.CodeDom.Compiler.GeneratedCodeAttribute("System.Runtime
    [System.Runtime.Serialization.DataContractAttribute(Name="User"
    [System.SerializableAttribute()]
    public partial class User : object, System.Runtime.Serializatio
{
....
}
```

从上面代码标红的部分可以看出，服务中定义的User只应用了DataContractAttribute属性，但生成的客户端User类中多了一个SerializableAtttribute。对于SerializableAttribute属性的作用与DataContract的作用是一样的，都是标记为该类支持序列化。因为在默认情况下，用户定义的类型并不支持序列化，只有应用了SerializableAttribute或DataContractAttribute属性的，.NET序列化器才能对该类型进行序列化。然而这两者又存在不同，Serializable要求它的所有程序都要支持序列化，如果发现不支持序列化的成员就会抛出异常，即Serializable会把类型的所有成员都进行序列化，如果想某个成员不序列化化，则必须显式标记NoSerialized属性；而DataContract却不同，标记了

DataContract属性的类只有标记了DataMember的成员才会被序列化，如果想类型的成员能够序列化，则应该应用DataMember属性。如果某个类型同时应用了DataContract和Serialized属性，如上面代码的User类，此时该类型将会只应用DataContract，即Serialized属性会忽略。我刚开始的疑问是，User类应用这两个属性，因为这两个属性对序列化成员有所区别，当时就纳闷到底是采取那个属性进行序列化的呢？经过查阅资料才发现了上面的结论，更多信息参考：Serialization in Windows Communication Foundation.aspx)。

然后利用该代理类实现对服务操作的调用，具体的实现代码如下所示：

```
namespace Client
{
    class Program
    {
        static void Main(string[] args)
        {
            UserValidationServiceClient wcfServiceProxy = new User\
            User newUser = new User() { UserName = "LearningHard",
            wcfServiceProxy.AddNewUser(newUser);

            // 演示已知类型的问题
            //Order order = new Order() { ID = Guid.NewGuid(), Date
            //wcfServiceProxy.AddOrder(order);

            // 获得用户信息
            string name = "Learning Hard Client";
            User user = wcfServiceProxy.GetUserByName(name);
            if (user != null)
            {
                Console.WriteLine("User Name is: " + user.UserName)
                Console.WriteLine("Email is: " + user.Email);
            }

            Console.WriteLine("Press any key to continue...");
            Console.Read();
        }
    }
}
```

经过上面的三步之后，我们就完成了WCF数据契约的实现。对于服务契约的调用过程是：客户端把相关需要序列化的对象序列化成XML格式，这里的格式与绑定的协议有关，因为上面设置的传输协议为http，所以这里应该序列化成XML格式的数据，然后再通过Http协议进行网络传递到服务，服务程序接收到传输过来的XML格式的数据，则利用DataContractSerializer反序列成User对象作为参数传递给AddNewUser方法；接着服务再把处理的后结果序列化成XML格式数据传递到客户端，客户端接收到服务程序响应的消息再进行反序列成具体的对象类型。对于操作GetUserByName的调用也是类似的。具体的运行结果如下图所示：



五、已知类型（KnownType）

因为WCF中使用DataContractSerializer进行序列化和反序列化的，由于DataContractSerializer进行序列化和反序列化时，都必须事先确定对象的类型。如果被序列化对象或反序列化生成的对象包含不可知的类型，序列化或反序列化将失败。所以为了保证DataContractSerializer正常的序列化和反序列化，需要将“未知”类型加入DataContractSerializer“已知”类型列表中。例如下面的服务契约：

```
// 服务契约
    [ServiceContract]
    //[ServiceKnownType(typeof(Order))] // 这是为了演示WCF已知类型
    public interface IUserValidationService
    {
        // 为了演示已知类型的操作方法
        [OperationContract]
        [ServiceKnownType(typeof(Order))]
        bool AddOrder(OrderBase order);
    }
```

假如，客户端同时定义了一个Order类：

以下代码能够成功通过编译，但在运行时却会失败：

原因在于我们并没有实际传递对象的引用，而是传递的是对象的XML结构。在上面的例子中，当我们传递的是Order对象而不是OrderBase对象时，服务并不知道它应该反序列为Order对象。

对于上面问题的解决办法就是让DataContractSerializer能够识别Order类型，成为DataContractSerializer的已知类型（Known Type）。DataContractSerializer内部具有一个已知类型的列表，我们需要将Order类型添加到这个列表中。对于已知类型，可以通过两个特性设置：KnownTypeAttribute和ServiceKnownTypeAttribute。KnownTypeAttribute应用于数据契约中，用于设置继承于该数据契约类型的子数据契约，或引用其他的契约类型。ServiceKnownTypeAttribute既可以应用于服务契约的接口和方法上，还可以应用在服务实现的类和方法上，应用在不同的目标元素，决定了定义已知类型的作用范围，下面，通过在基类OrderBase指定了子契约的类型Order：

而ServiceKnownTypeAttribute特性，不仅可以使用在服务契约类型上，还可以应用在服务契约的操作方法上。如果应用在服务契约类型上，则已知类型在所有实现了该契约的服务操作中都有效，即作用范围为服务契约界别的，如果应用于服务契约的操作方法上，则定义的已知类型仅在实现了该契约的服务操作中有效。

```
  // 服务契约
    [ServiceContract]
    [ServiceKnownType(typeof(Order))]  // 服务契约级别
    public interface IUserValidationService
    {
        // 为了演示已知类型的操作方法
        //[OperationContract]
        //[ServiceKnownType(typeof(Order))] // 单个服务操作级别
        bool AddOrder(OrderBase order);
    }
```

除了通过特性的方式设置已知类型外，还可以通过配置文件的方式来进行指定。已知类型定义在<System.runtime.serialization>配置节点中，可以采用下面的方式来定义：

```
<configuration>
  <system.runtime.serialization>
    <dataContractSerializer>
      <declaredTypes>
        <add type="BusinessEntity.OrderBase,BusinessEntity.KnownTyp
          <knownType type="BusinessEntity.Order,BusinessEntity.Know
        </add>
      </declaredTypes>
    </dataContractSerializer>
  </system.runtime.serialization>
</configuration>
```

六、总结

到这里，数据契约的分享就结束。对于这篇博文首先介绍了数据契约和序列化的基本知识，接着介绍了.NET中的序列化机制和WCF中序列化机制，最后完成了一个数据契约的例子。看完本篇文章应该明确几个问题：

1. SerializableAttribute与DataContract异同。

答：相同点：都是标记类型为可序列化类型

不同点：在于序列化的成员不一样，DataContract是Opt-in(明确参与)的方式，即使用DataMember特性明确标识哪些成员需要序列化，而Serializable是Opt-out方式，即使用NoSerializable特性明确标识不参与序列化的成员。

2. BinartFormatter、DataContractSerializer和XmlSerializer的区别，具体答案见下图和参考下面博文：XmlSerializer, DataContractSerializer 和 BinaryFormatter区别与用法分析。

Figure 1 Comparing Serializers

Feature	XmlSerializer	DataContractSerializer NetDataContractSerializer
Explicitness	Opt-out	Opt-in * Opt-out **
Default mapping	Public fields/props	All [DataMember]s * All fields **
Attribute required	No	Yes
Default order	Same as class	Alphabetical
XML Schema	Extensive	Constrained
Code generator	Xsd.exe	SvcUtil.exe
Override	IXmlSerializable	ISerializable
Type fidelity	No	NetDataContractSerializer
Versioning support	No	Yes
Initialization	Constructor	Callbacks
Compatibility	ASMX	.NET Remoting

好的博文记录：Create and Consume RESTFul Service in .NET Framework 4.0

本文所有源代码下载：WCFDataContract.zip

跟我一起学WCF(8)——WCF中Session、实例管理详解

一、引言

由前面几篇博文我们知道，WCF是微软基于SOA建立的一套在分布式环境中各个相对独立的应用进行交流（Communication）的框架，它实现了最新的基于WS-*规范。按照SOA的原则，相对独自的业务逻辑以Service的形式进行封装，调用者通过消息(Messaging)的方式来调用服务。对于承载某个业务功能实现的服务应该具有上下文(Context)无关性，意思就是说构造服务的操作(Operation)不应该绑定到具体的调用上下文，对于任何的调用，具有什么的样输入就会对应怎样的输出。因为SOA一个最大的目标是尽可能地实现重用，只有具有Context无关性，服务才能最大限度的重用。即从软件架构角度理解为，一个模块只有尽可能的独立，即具有上下文无关性，才能被最大限度的重用。软件体系一直在强调低耦合也是这个道理。

但是在某些场景下，我们却希望系统为我们创建一个Session来保留Client和Service的交互的状态，如Asp.net中Session的机制一样，WCF也提供了对Session的支持。下面就具体看看WCF中对Session的实现。

二、WCF中Session详细介绍

2.1 Asp.net的Session与WCF中的Session

在WCF中，Session属于Service Contract的范畴，并在Service Contract定义中通过SessionModel.aspx)参数来实现。WCF中会话具有以下几个重要的特征：

- Session都是由Client端显示启动和终止的。

在WCF中Client通过创建的代理对象来和服务进行交互，在支持Session的默认情况下，Session是和具体的代理对象绑定在一起，当Client通过调用代理对象的某个方法来访问服务时，Session就被初始化，直到代理的关闭，Session则被终止。我们可以通过两种方式来关闭Proxy:一是调用ICommunicationObject.Close 方法.aspx)，二是调用ClientBase<TChannel>.Close 方法.aspx) 。我们也可以通过服务中的某个操作方法来初始化、或者终止Session，可以通过OperationContractAttribute的IsInitiating和IsTerminating.aspx)参数来指定初始化和终止Session的Operation。

- 在WCF会话期间，传递的消息按照它发送的顺序被接收。
- WCF并没有为Session的支持保存相关的状态数据。

讲到Session，做过Asp.net开发的人，自然想到的就是Asp.net中的Session。它们只是名字一样，在实现机制上有很大的不同。Asp.net中的Session具有以下特性：

- Asp.net的Session总是由服务端启动的，即在服务端进行初始化的。

- Asp.net中的Session是无需，不能保证请求处理是有序的。
- Asp.net是通过在服务端以某种方式保存State数据来实现对Session的支持，例如保存在Web Server端的内存中。

2.2 WCF中服务实例管理

对于Client来说，它实际上不能和Service进行直接交互，它只能通过客户端创建的Proxy来间接地和Service进行交互，然而真正的调用而是通过服务实例来进行的。我们把通过Client的调用来创建最终的服务实例过程称作激活，在.NET Remoting中包括Singleton模式、SingleCall模式和客户端激活方式，WCF中也有类似的服务激活方式：单调服务（PerCall）、会话服务（PerSession）和单例服务（Singleton）。

- 单调服务（**Percall**）：为每个客户端请求分配一个新的服务实例。类似.NET Remoting中的SingleCall模式
- 会话服务（**Persession**）：在会话期间，为每次客户端请求共享一个服务实例，类似.NET Remoting中的客户端激活模式。
- 单例服务（**Singleton**）：所有客户端请求都共享一个相同的服务实例，类似于.NET Remoting的Singleton模式。但它的激活方式需要注意一点：当为对于的服务类型进行Host的时候，与之对应的服务实例就被创建出来，之后所有的服务调用都由这个服务实例进行处理。

WCF中服务激活的默认方式是PerSession，但不是所有的Bingding都支持Session，比如BasicHttpBinding就不支持Session。你也可以通过下面的方式使ServiceContract不支持Session.

```
[ServiceContract(SessionMode = SessionMode.NotAllowed)]
```

下面分别介绍下这三种激活方式的实现。

三、WCF中实例管理的实现

WCF中服务激活的默认是PerSession的方式，下面就看看PerSession的实现方式。我们还是按照前面几篇博文的方式来实现使用PerSession方式的WCF服务程序。

第一步：自然是实现我们的WCF契约和契约的服务实现。具体的实现代码如下所示：

```
 1 // 服务契约的定义
 2     [ServiceContract]
 3     public interface ICalculator
 4     {
 5         [OperationContract(IsOneWay = true)]
 6         void Increase();
 7 
 8         [OperationContract]
 9         int GetResult();
10     }
11 
12   // 契约的实现
13     public class CalculatorService : ICalculator, IDisposable
14     {
15         private int _nCount = 0;
16 
17         public CalculatorService()
18         {
19             Console.WriteLine("CalulatorService object has been
20         }
21 
22         // 为了看出服务实例的释放情况
23         public void Dispose()
24         {
25             Console.WriteLine("CalulatorService object has been
26         }
27 
28         #region ICalulator Members
29         public void Increase()
30         {
31             // 输出Session ID
32             Console.WriteLine("The Add method is invoked and the
33             this._nCount++;
34         }
35 
36         public int GetResult()
37         {
38             Console.WriteLine("The GetResult method is invoked a
39             return this._nCount;
40         }
41         #endregion
42     }
```

为了让大家对服务对象的创建和释放有一个直观的认识，我特意对服务类实现了构造函数和IDisposable接口，同时在每个操作中输出当前的Session ID。

第二步：实现服务宿主程序。这里还是采用控制台程序作为服务宿主程序，具体的实现代码如下所示：

```
1 // 服务宿主程序
2     class Program
3     {
4         static void Main(string[] args)
5         {
6             using (ServiceHost host = new ServiceHost(typeof(Ca
7             {
8                 host.Opened += delegate
9                 {
10                    Console.WriteLine("The Calculator Service ha
11                };
12
13                host.Open();
14                Console.ReadLine();
15            }
16        }
17    }
```

对应的配置文件为：

```
<!--服务宿主的配置文件-->
<configuration>
  <system.serviceModel>
      <behaviors>
          <serviceBehaviors>
              <behavior name ="CalculatorBehavior">
                  <serviceMetadata httpGetEnabled="true"/>
              </behavior>
          </serviceBehaviors>
      </behaviors>
  <services>
      <service name="WCFContractAndService.CalculatorService" beha
          <endpoint address="" binding="basicHttpBinding" contract=
      <host>
          <baseAddresses>
              <add baseAddress="http://localhost:9003/CalculatorPe
          </baseAddresses>
      </host>
      </service>
  </services>
  </system.serviceModel>
</configuration>
```

第三步：实现完了服务宿主程序，接下来自然是实现客户端程序来访问服务操作。这里的客户端也是控制台程序，具体的实现代码如下所示：

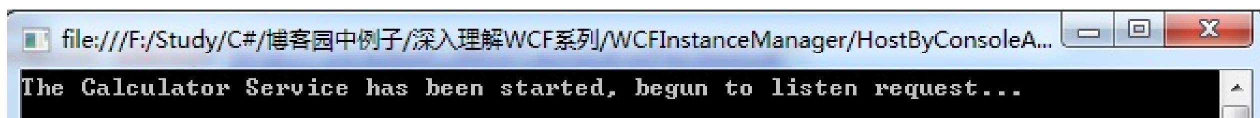
```
 1 // 客户端程序实现
 2     class Program
 3     {
 4         static void Main(string[] args)
 5         {
 6             // Use ChannelFactory<ICalculator> to create WCF Se
 7             ChannelFactory<ICalculator> calculatorChannelFactory
 8             Console.WriteLine("Create a calculator proxy :proxy
 9             ICalculator proxy1 = calculatorChannelFactory.Create
10             Console.WriteLine("Invoke proxy1.Increate() method")
11             proxy1.Increase();
12             Console.WriteLine("Invoke proxy1.Increate() method a
13             proxy1.Increase();
14             Console.WriteLine("The result return via proxy1.GetR
15 
16             Console.WriteLine("Create another calculator proxy: 
17             ICalculator proxy2 = calculatorChannelFactory.Create
18             Console.WriteLine("Invoke proxy2.Increate() method")
19             proxy2.Increase();
20             Console.WriteLine("Invoke proxy2.Increate() method a
21             proxy2.Increase();
22             Console.WriteLine("The result return via proxy2.GetR
23 
24             Console.ReadLine();
25         }
26     }
```

客户端对应的配置文件内容如下所示：

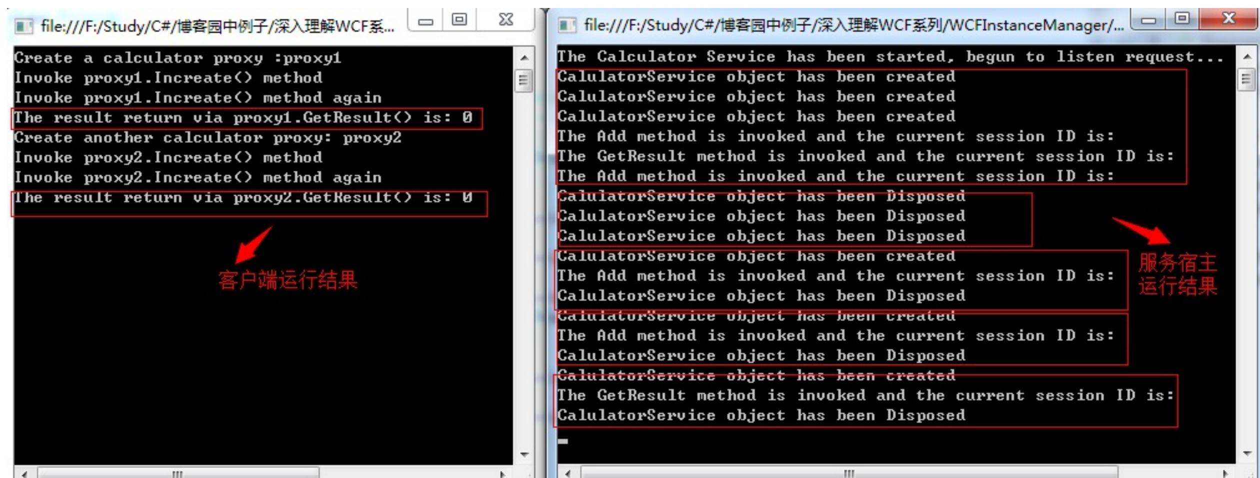
```
<!--客户端配置文件-->
<configuration>
    <system.serviceModel>
        <client>
            <endpoint address="http://localhost:9003/CalculatorPerS
                      binding="basicHttpBinding"
                      contract="WCFContractAndService.ICalculator"
        </client>
    </system.serviceModel>
</configuration>
```

经过上面三步，我们就完成了PerSession方式的WCF程序了，下面看看该程序的运行结果。

首先，以管理员权限运行服务寄宿程序，运行成功后，你将看到如下图所示的画面：



接下来，运行客户端对服务操作进行调用，运行成功后，你将看到服务宿主的输出和客户端的输出情况如下图所示：



从客户端的运行结果可以看出，虽然我们两次调用了Increase方法来增加_nCount的值，但是最终的运行结果仍然是0。这样的运行结果好像与我们之前所说的WCF默认Session支持矛盾，因为如果WCF默认的方式PerSession的话，则服务实例是和Proxy绑定在一起，当Proxy调用任何一个操作的时候Session开始，从此Session将会与Proxy具有一样的生命周期。按照这个描述，客户端运行的结果应该是2而不是0。这里，我只能说运行结果并没有错，因为有图有真相嘛，那到底是什么原因导致客户端获得_nCount值是0呢？其实在前面已经讲到过，并不是所有的绑定都是支持Session的，上面程序的实现我们使用的basicHttpBinding，而basicHttpBinding是不支持Session方式的，所以WCF会采用PerCall的方式创建Service Instance，所以在服务端中对于每一个Proxy都有3个对象被创建，两个是对Increase方法的调用会导致服务实例的激活，另一个是对GetResult方法的调用导致服务实例的激活。因为是PerCall方式，所以每次调用完之后，就会对服务实例进行释放，所以对应的就有3行服务对象释放输出。并且由于使用的是不支持Session的binding，所以Session ID的输出也为null。所以，上面WCF程序其实是PerCall方式的实现。

既然，上面的运行结果是由于使用了不支持Session的basicHttpBinding导致的，下面看看使用一个支持Session的Binding：wsHttpBinding来看看运行结果是怎样的，这里的修改很简单，只需要把宿主和客户端的配置文件把绑定类型修改为wsHttpBinding就可以了。

```
<!--客户端配置文件-->
<configuration>
    <system.serviceModel>
        <client>
            <endpoint address="http://localhost:9003/CalculatorPerS
                      binding="**wsHttpBinding**"
                      contract="WCFContractAndService.ICalculator"
        </client>
    </system.serviceModel>
</configuration>
<!--服务宿主的配置文件-->
<configuration>
  <system.serviceModel>
      <behaviors>
          <serviceBehaviors>
              <behavior name ="CalculatorBehavior">
                  <serviceMetadata httpGetEnabled="true"/>
              </behavior>
          </serviceBehaviors>
      </behaviors>
  <services>
      <service name="WCFContractAndService.CalculatorService" behav
          <endpoint address="" binding="**wsHttpBinding**" contract
      <host>
          <baseAddresses>
              <add baseAddress="http://localhost:9003/CalculatorPer
          </baseAddresses>
      </host>
      </service>
  </services>
  </system.serviceModel>
</configuration>
```

现在我们再运行下上面的程序来看看此时的执行结果，具体的运行结果如下图所示：

```
file:///F:/Study/C#/博客园中例子/深入理解WCF系...
Create a calculator proxy :proxy1
Invoke proxy1.Increate() method
Invoke proxy1.Increate() method again
The result return via proxy1.GetResult() is: 2
Create another calculator proxy: proxy2
Invoke proxy2.Increate() method
Invoke proxy2.Increate() method again
The result return via proxy2.GetResult() is: 2
```

```
file:///F:/Study/C#/博客园中例子/深入理解WCF系列/WCFInstanceManager/HostByConsoleA...
The Calculator Service has been started, begun to listen request...
CalulatorService object has been created
The Add method is invoked and the current session ID is: urn:uuid:f9a39fdc-6c34-4e2a-b804-24e513b54c92
The Add method is invoked and the current session ID is: urn:uuid:f9a39fdc-6c34-4e2a-b804-24e513b54c92
The GetResult method is invoked and the current session ID is: urn:uuid:f9a39fdc-6c34-4e2a-b804-24e513b54c92
CalulatorService object has been created
The Add method is invoked and the current session ID is: urn:uuid:d2a9b64f-8f95-4929-bcad-f7c91b15b122
The Add method is invoked and the current session ID is: urn:uuid:d2a9b64f-8f95-4929-bcad-f7c91b15b122
The GetResult method is invoked and the current session ID is: urn:uuid:d2a9b64f-8f95-4929-bcad-f7c91b15b122
```

从上面的运行结果可以看出，此时两个Proxy的运行结果都是2，可以看出此时服务激活方式采用的是PerSession方式。此时对于服务端就只有两个服务实例被创建了，并且对于每个服务实例具有相同的Session ID。 另外由于Client的Proxy还依然

存在，服务实例也不会被回收掉，从上面服务端运行的结果也可以证实这点，因为运行结果中没有对象呗Disposable的输出。你可以在客户端显式调用ICommunicationObject.Close方法来显式关闭掉Proxy，在客户端添加对Proxy的显示关闭代码，此时客户端的代码修改为如下所示：

```
 1 // 客户端程序实现
 2     class Program
 3     {
 4         static void Main(string[] args)
 5         {
 6             // Use ChannelFactory<ICalculator> to create WCF Ser
 7             ChannelFactory<ICalculator> calculatorChannelFactory
 8             Console.WriteLine("Create a calculator proxy :proxy1
 9             ICalculator proxy1 = calculatorChannelFactory.Create
10             Console.WriteLine("Invoke proxy1.Increate() method")
11             proxy1.Increase();
12             Console.WriteLine("Invoke proxy1.Increate() method a
13             proxy1.Increase();
14             Console.WriteLine("The result return via proxy1.GetR
15             **(proxy1 as ICommunicationObject).Close(); // 显示关
16 
17             Console.WriteLine("Create another calculator proxy:
18             ICalculator proxy2 = calculatorChannelFactory.Create
19             Console.WriteLine("Invoke proxy2.Increate() method")
20             proxy2.Increase();
21             Console.WriteLine("Invoke proxy2.Increate() method a
22             proxy2.Increase();
23             Console.WriteLine("The result return via proxy2.GetR
24             **(proxy2 as ICommunicationObject).Close();** 25
26             Console.ReadLine();
27         }
28     }
```

此时，服务对象的Dispose()方法将会调用，此时服务端的运行结果如下图所示：

```
file:///F:/Study/C#/博客园中例子/深入理解WCF系列/WCFInstanceManager/HostByConsoleA...
The Calculator Service has been started, begun to listen request...
CalulatorService object has been created
The Add method is invoked and the current session ID is: urn:uuid:4b1698f8-815e-
4cbb-9e2d-b53cf944fcfa
The Add method is invoked and the current session ID is: urn:uuid:4b1698f8-815e-
4cbb-9e2d-b53cf944fcfa
The GetResult method is invoked and the current session ID is: urn:uuid:4b1698f8
-815e-4cbb-9e2d-b53cf944fcfa
CalulatorService object has been Disposed
CalulatorService object has been created
The Add method is invoked and the current session ID is: urn:uuid:57facbdc-4468-
4605-8fcf-66e924013b93
The Add method is invoked and the current session ID is: urn:uuid:57facbdc-4468-
4605-8fcf-66e924013b93
The GetResult method is invoked and the current session ID is: urn:uuid:57facbdc
-4468-4605-8fcf-66e924013b93
CalulatorService object has been Disposed
```

上面演示了默认支持Session的情况，下面我们修改服务契约使之不支持Session，此时只需要知道ServiceContract的SessionMode为NotAllowed即可。

```
[ServiceContract**(SessionMode= SessionMode.NotAllowed)**] // 是服务
    public interface ICalculator
    {
        [OperationContract(IsOneWay = true)]
        void Increase();

        [OperationContract]
        int GetResult();
    }
```

此时，由于服务契约不支持Session，此时服务激活方式采用的仍然是PerCall。运行结果与前面采用不支持Session的绑定的运行结果一样，这里就不一一贴图了。

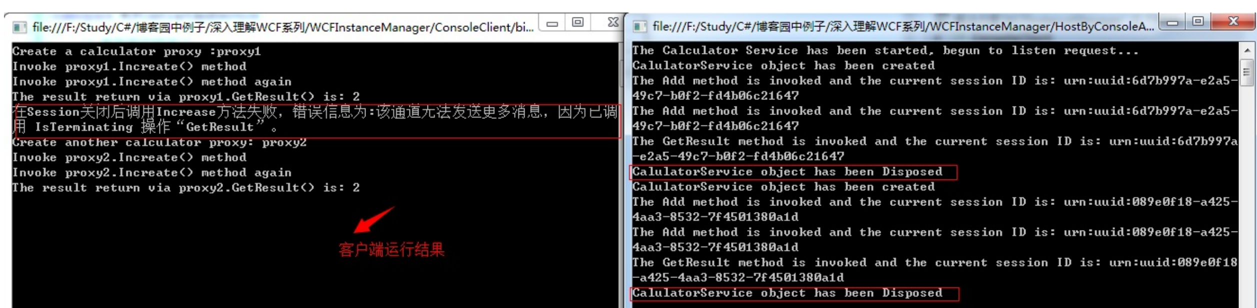
除了通过显式修改ServiceContract的SessionMode来使服务契约支持或不支持Session外，还可以定制操作对Session的支持。定制操作对Session的支持可以通过OperationContract的IsInitiating和InTerminating属性设置。

```
1  // 服务契约的定义
2     [ServiceContract(SessionMode= SessionMode.Required)] // 显式
3     public interface ICalculator
4     {
5         // IsInitiating:该值指示方法是否实现可在服务器上启动会话（如果存
6         // IsTerminating:获取或设置一个值，该值指示服务操作在发送答复消
7         [OperationContract(IsOneWay = true, IsInitiating =true,
8         void Increase();
9
10        [OperationContract(IsInitiating = true, IsTerminating =
11        int GetResult();
12    }
```

在上面代码中，对两个操作都设置InInitiating的属性为true，意味着调用这两个操作都会启动会话，而把GetResult操作的IsTerminating设置为true，意味着调用完这个操作后，会导致服务关闭掉会话，因为在Session方式下，Proxy与Session有一致的生命周期，所以关闭Session也就是关闭proxy对象，所以如果后面再对proxy对象的任何一个方法进行调用将会导致异常，下面代码即演示了这种情况。

```
1 // 客户端程序实现
2     class Program
3     {
4         static void Main(string[] args)
5         {
6             // Use ChannelFactory<ICalculator> to create WCF Se
7             ChannelFactory<ICalculator> calculatorChannelFactory
8             Console.WriteLine("Create a calculator proxy :proxy
9             ICalculator proxy1 = calculatorChannelFactory.Create
10            Console.WriteLine("Invoke proxy1.Increate() method")
11            proxy1.Increase();
12            Console.WriteLine("Invoke proxy1.Increate() method a
13            proxy1.Increase();
14            Console.WriteLine("The result return via proxy1.GetR
15            **try
16            {
17                proxy1.Increase(); // session关闭后对proxy1.Incre
18            }
19            catch (Exception ex) // 异常捕获
20            {
21                Console.WriteLine("在Session关闭后调用Increase方法
22            }** 23
24            Console.WriteLine("Create another calculator proxy:
25            ICalculator proxy2 = calculatorChannelFactory.Create
26            Console.WriteLine("Invoke proxy2.Increate() method")
27            proxy2.Increase();
28            Console.WriteLine("Invoke proxy2.Increate() method a
29            proxy2.Increase();
30            Console.WriteLine("The result return via proxy2.GetR
31
32            Console.ReadLine();
33        }
34    }
```

此时运行结果也验证我们上面的分析，客户端和服务端的运行结果如下图所示：



上面演示了PerSession和PerCall的两种服务对象激活方式，下面看看Single的激活方式运行的结果。首先通过ServiceBehavior的InstanceContextMode属性显式指定激活方式为Single，由于ServiceBehaviorAttribute.aspx)特性只能应用于类上，所以把该特性应用于CalculatorService类上，此时服务实现的代码如下所示：

```
// 契约的实现
    // ServiceBehavior属性只能应用在类上
    **[ServiceBehavior(InstanceContextMode = InstanceContextMode.Si
    public class CalculatorService : ICalculator, IDisposable
    {
        private int _nCount = 0;

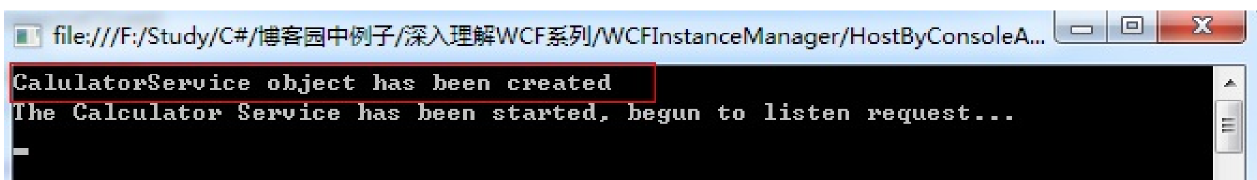
        public CalculatorService()
        {
            Console.WriteLine("CalulatorService object has been cre
        }

        // 为了看出服务实例的释放情况
        public void Dispose()
        {
            Console.WriteLine("CalulatorService object has been Dis
        }

        #region ICalulator Members
        public void Increase()
        {
            // 输出Session ID
            Console.WriteLine("The Add method is invoked and the cu
            this._nCount++;
        }

        public int GetResult()
        {
            Console.WriteLine("The GetResult method is invoked and
            return this._nCount;
        }
        #endregion
    }
```

此时运行服务宿主的输出结果如下图所示：



从运行结果可以看出，对于Single方式，服务实例在服务类型被寄宿的时候就已经创建了，对于PerCall和PerSession方式而是在通过Proxy调用相应的服务操作之后，服务实例才开始创建的。下面运行客户端程序，你将看到如下图所示的运行结果：

此时，第二个Proxy返回的结果是4而不是2，这是因为采用Single方式只存在一个服务实例，所有的调用状态都将保留，所以_nCount的值在原来的基础上继续累加。

四、总结

到这里，本文的分享就结束了，本文主要分享了WCF中实例管理的实现。从WCF的实例实现可以看出，WCF实例实现是借鉴了.NET Remoting中实例实现，然后分别分享了服务实例三种激活方式在WCF中的实现，并通过对运行结果进行对比来让大家理解它们之间的区别。

本文所以源码：WCFInstanceManager.zip

跟我一起学WCF(9)——WCF回调操作的实现

一、引言

在上一篇文章中介绍了WCF对Session的支持，在这篇文章中将详细介绍WCF支持的操作。在WCF中，除了支持经典的请求/应答模式外，还提供了对单向操作、双向回调操作模式的支持，此外还有流操作的支持。接下来将详细介绍下这几种操作，并实现一个双向回调操作的例子。

二、WCF操作详解

2.1 请求—应答操作

请求应答模式是WCF中默认的操作模式。请求应答模式指的是：客户端以消息形式发送请求，它会阻塞客户端直到收到应答消息。应答的默认超时时间为1分钟，如果超过这一时间服务仍然没有应答，客户端就会获得一个TimeOutException异常。WCF中除了NetPeerTcpBinding和NetMsmqBinding绑定，所有的绑定都支持请求—应答操作。

2.2 单向操作

单向操作是没有返回值的，客户端不关心调用是否成功。单向操作指的是：客户端一旦发出调用请求，WCF会生成一个请求消息发送给服务端，但客户端并不需要接收相关的应答消息。因此，单向操作不能有返回值，并且服务端抛出的任何异常都不会传递给客户端。所以客户端如果需要捕获服务端发生的异常，此时不能把操作契约的IsOneWay属性设置为true，该属性的默认值为false。异常处理参考：如何在WCF进行Exception Handling。单向操作不等同于异步操作，单向操作只是在发出调用的瞬间阻塞客户端，但如果发出多个单向调用，WCF会将请求调用放入服务端的队列中，并在某个时间进行执行。队列的存储个数有限，一旦发出的调用个数超出了队列容量，则会发生阻塞现象，此时调用请求无法放入丁烈，直到有其他请求被处理，即队列中的请求出队列后，产生阻塞的调用就会放入队列，并解除对客户端的阻塞。WCF中所有绑定都支持单向操作。WCF中实现单向操作只需要设置IsOneWay属性为true即可。这里需要注意一点：由于单向操作没有应答消息，因此它不能包含返回结果。

2.3 回调操作

WCF支持服务将调用返回给它的客户端。在回调期间，服务成为了客户端，而客户端成为了服务。在WCF中，并不是所有的绑定都支持回调操作，只有具有双向能力的绑定才能够用于回调。例如，HTTP本质上是与连接无关的，所以它不能用于回

调，因此我们不能基于basicHttpBinding和wsHttpBinding绑定使用回调，WCF为NetTcpBinding和NetNamedPipeBinding提供了对回调的支持，因为TCP和IPC协议都支持双向通信。为了让Http支持回调，WCF提供了WsDualHttpBinding绑定，它实际上设置了两个Http通道：一个用于从客户端到服务的调用，另一个用于服务到客户端的调用。

回调操作时通过回调契约来实现的，回调契约属于服务契约的一部分，一个服务契约最多只能包含一个回调契约。一旦定义了回调契约，就需要客户端实现回调契约。在WCF中，可以通过ServiceContract的CallbackContract.aspx)属性来定义回调契约。具体的实现代码如下所示：

```
// 指定回调契约为ICallback
    [ServiceContract(Namespace="http://cnblog.com/zhili/", Callback
    public interface ICalculator
    {
        [OperationContract(IsOneWay = true)]
        void Multiple(double a, double b);
    }

// 回调契约的定义，此时回调契约不需要应用ServiceContractAttribute特性
    public interface ICallback
    {
        [OperationContract(IsOneWay = true)]
        void DisplayResult(double x, double y, double result);
    }
```

在上面代码中，回调契约不必标记ServiceContract特性，因为类型只要被定义为回调契约，就代表它具有ServiceContract特性，但仍然需要为所有的回调接口中的方法标记OperationContract特性。

2.4 流操作

在默认情况下，当客户端与服务交换消息时，这些消息会被放入到接收端的缓存中，一旦接收到完整的消息，就立即被传递处理。无论是客户端发送消息到服务还是服务返回消息给客户端，都是如此。当客户端调用服务时，只要接收到完整的消息，服务就会被调用，当包含了调用结果的返回消息被客户端完整接收时，才会接触对客户端的阻塞。对于数据量小的消息，这种交换模式提供了简单的编程模型，因为接收消息的耗时较短，然而，一旦处理数据量更大的消息，例如包含了多媒体内容或大文件，如果每次都要等到完整地接收消息之后才能解除阻塞，这未免也不现实。为了解决这样的问题，WCF允许接收端通过通道接收消息的同时，启动对消息数据的处理，这样的处理过程称为流传输模型。对于具有大量负载的消息而言，流操作改善了系统的吞吐量和响应速度，因为在发生和接收消息的同时，不管是发送端还是接收端都不会被阻塞。

三、WCF中回调操作的实现

上面介绍了WCF中支持的四种操作，下面就具体看看WCF中回调操作的实现。该例子的基本原理是：客户端调用服务操作，服务操作通过客户端上下文实例调用客户端操作。下面还是按照三个步骤来实现该WCF程序。

第一步：同样是实现WCF服务契约和契约的实现。具体的实现代码如下所示：

```
 1 // 指定回调契约为ICallback
 2     [ServiceContract(Namespace="http://cnblog.com/zhili/", Callb
 3     public interface ICalculator
 4     {
 5         [OperationContract(IsOneWay = true)]
 6         void Multiple(double a, double b);
 7     }
 8 
 9  // 回调契约的定义，此时回调契约不需要应用ServiceContractAttribute特性
10     public interface ICallback
11     {
12         [OperationContract(IsOneWay = true)]
13         void DisplayResult(double x, double y, double result);
14     }
15 
16  // 服务契约的实现
17     public class CalculatorService : ICalculator
18     {
19         #region ICalculator Members
20         public void Multiple(double a, double b)
21         {
22             double result = a * b;
23             // 通过客户端实例通道
24             ICallback callback = OperationContext.Current.GetCal
25 
26             // 对客户端操作进行回调
27             callback.DisplayResult(a, b, result);
28         }
29         #endregion
30     }
```

第二步：实现服务宿主。这里还是以控制台程序作为服务宿主。具体的实现代码如下所示：

```
 1 class Program
 2     {
 3         static void Main(string[] args)
 4         {
 5             using (ServiceHost host = new ServiceHost(typeof(Cal
 6             {
 7                 host.Opened += delegate
 8                 {
 9                     Console.WriteLine("Service start now....");
10                 };
11 
12                 host.Open();
13                 Console.Read();
14             }
15         }
16     }
```

宿主对应的配置文件内容如下所示：

```
<configuration>
    <system.serviceModel>
        <behaviors>
            <serviceBehaviors>
                <behavior>
                    <serviceMetadata httpGetEnabled="true" httpGetU
                </behavior>
            </serviceBehaviors>
        </behaviors>
        <services>
            <service name="WCFContractAndService.CalculatorService"
                <endpoint address="net.tcp://localhost:9003/Calcula
            </service>
        </services>
    </system.serviceModel>
</configuration>
```

第三步：实现客户端。由于服务端来对客户端操作进行回调，所以此时客户端需要实现回调契约。首先以管理员权限启动服务宿主，服务宿主启动成功之后，客户端通过添加服务引用的方式来生成客户端代理类，此时需要在添加服务引用窗口地址中输入：http://localhost:8080/Metadata。添加服务引用成功之后，接着在客户端实现回调契约，具体的实现代码如下所示：

```
1 // 客户端中对回调契约的实现
2     public class CallbackWCFService : ICalculatorCallback
3     {
4         public void DisplayResult(double a, double b, double resu
5         {
6             Console.WriteLine("{0} * {1} = {2}", a, b, result);
7         }
8     }
```

接下来就是实现测试回调操作的客户端代码了。具体的实现步骤是：实例化一个回调类的实例，然后把它作为上下文实例的操作，最后把上下文实例作为客户端代理的参数来实例化客户端代理。具体的实现代码如下所示：

```
 1  // 客户端实现，测试回调操作
 2     class Program
 3     {
 4         static void Main(string[] args)
 5         {
 6             InstanceContext instanceContex = new InstanceContext
 7             CalculatorClient proxy = new CalculatorClient(instan
 8             proxy.Multiple(2,3);
 9
10             Console.Read();
11         }
12     }
```

下面运行运行该程序来检测下该程序是否能够成功回调，首先以管理员权限启动服务宿主程序，再启动客户端程序，如果回调成功，你将看到如下图所示的运行结果：



这里只是演示了回调操作的实现，关于流操作的实现，这里就不再去实现了，等具体需要的时候再去研究吧，同时给出关于流操作实现的参考文章：

Stream Operation in WCF

WCF流处理（Streaming）机制

四、总结

到这里，WCF操作的内容就分享结束了，本文首先介绍了在WCF中支持四种操作：请求-应答操作、单向操作、回调操作和流操作，WCF中默认的操作时请求-应答操作，最后实现了一个回调操作的实例。

本文所有源码：WCFCallbackOperation.zip

跟我一起学WCF(10)——WCF中事务处理

一、引言

好久没更新，总感觉自己欠了什么一样的，所以今天迫不及待地来更新了，因为后面还有好几个系列准备些，还有很多东西需要学习总结的。今天就来介绍下WCF对事务的支持。

二、WCF事务详解

2.1 事务概念与属性

首先，大家在学习数据库的时候就已经接触到事务这个概念了。所谓事务，它是一个操作序列，这些操作要么都执行，要么都不执行，它是一个不可分割的工作单元。例如，银行转账功能，这个功能涉及两个逻辑操作

1. 从一个账户A中扣钱
2. 另一个账户B增加对应的钱。

现实生活中，这两个操作需要要么都执行，要么都不执行。所以在实现转账功能时，这两个操作就可以作为一个事务来进行提交，这样才能够保证转账功能的正确执行。

上面通过银行转账的例子来解释了事务的概念了，也可以说非常容易理解。然后在数据库的相关书籍里面都会介绍事务的特性。一个逻辑工作单元要成为事务，必须满足四个特性，这四个特性包括原子性、一致性、隔离性和持久性。这四个特性也简称为ACID（ACID是四个特性英文单词首字母的缩写）。

- 原子性。此属性可确保特定事务下完成的所有更新都已提交并保持持久，或所有这些更新都已中止并回滚到其先前状态。
- 一致性。此属性可保证某一事务下所做的更改表示从一致状态转换到另一种一致状态。例如，将钱从支票帐户转移到存款帐户的事务并不改变整个银行帐户中的钱的总额。
- 隔离。此属性可防止事务遵循属于其他并发事务的未提交的更改。隔离在确保一种事务不能对另一事务的执行产生意外的影响的同时，还提供一个抽象的并发。
- 持久性。这意味着一旦提交对托管资源（如数据库记录）的更新，即使出现失败这些更新也会保持持久。

2.2 事务协议

WCF支持分布式事务，也就是说WCF中的事务可以跨越服务边界、进程、机器和网络，在多个客户端和服务之间存在。即WCF中事务可以被传播的。既然WCF支持事务，则自然就有对应传输事务信息的相关协议。所以也就有了事务协议。

WCF使用不同的事务协议来控制事务的执行范围，事务协议是为了实现分布式环境中事务的传播。WCF支持以下三种事务协议：

1. 轻量级协议（Lightweight Protocol）：该协议仅用于管理本地环境中的事务，即处于同一应用程序域中的事务。它无法跨越应用程序域的边界传播事务，则更不用说跨越进程和机器边界了。所以Lightweight Protocol只适用于某个服务的内部或外部。但相对于其他协议来说，轻量级协议的性能是最好的，这应该是显然的，不能跨越进程和机器边界，则就不存在网络传输。
2. OleTx协议：该协议用于跨应用程序域、进程和机器边界传播事务。协议采用远程过程调用（RPC），并采用Windows专用的二进制格式。但该协议无法穿越防火墙或与非Windows方协作。所以OleTx协议多用于Windows体系下的内网环境（即Intranet环境）。
3. WS-Atomic(WS原子性，WSAT)事务协议：该协议与OleTx协议类似，同样允许事务穿越应用程序域、进程和机器边界传播事务。但不同于OleTx协议的是，WSAT协议基于一种行业标准，它使用HTTP协议，并编码形式为文本格式，因而可以穿越防火墙。虽然可以在内网中使用WSAT协议，但它主要还是用于Internet环境。

因为轻量级协议不能跨越服务边界传播事务，所有没有绑定支持轻量级协议。WCF预定义的绑定中实现了标准的WS-Atomic 协议和Microsoft专有的OleTx协议，我们可以通过编程或配置文件来设置事务协议。具体设置方法如下所示：

```
 1 <bindings>
 2    <netTcpBinding>
 3         <!--通过transactionProtocol属性来设置事务协议-->
 4         <binding name="transactionTCP" transactionFlow="true" tr
 5       </netTcpBinding>
 6 </bindings>
 7 // 通过编程设置
 8  NetTcpBinding tcpBinding = new NetTcpBinding();
 9             // 注意： 事务协议的配置只有在事务传播的情况下才有意义
10             tcpBinding.TransactionFlow = true;
11             tcpBinding.TransactionProtocol = TransactionProtocol
```

这里需要注意，事务协议的配置只有在允许事务传播的情况下才有意义。并且NetTcpBinding和NetNamedPipeBinding都提供了TransactionProtocol属性。由于TCP和IPC绑定只能在内网使用，将它们设置为WSAT协议并无实际意义，对于WS绑定（如WSHttpBinding、WSDualHttpBinding和WSFederationHttpBinding）并没有TransactionProtocol属性，它们设计的目的在于当涉及多个使用WAST协议的事务管理器时，能够跨越Internet。但如果只有一个事务协调器，OleTx协议将是默认的协议，不必也不能为它配置一个特殊的协议。

2.3 事务管理器

分布式事务的实现要依靠第三方事务管理器来实现。它负责管理一个个事务的执行情况，最后根据全部事务的执行结果，决定提交或回滚整个事务。WCF提供了三个不同的事务管理器，它们分别是轻量级事务管理器（LTM）、核心事务管理器（KTM）和分布式事务协调器（DTC）。WCF根据平台使用的公共，应用程序的事务执行的任务、调用的服务以及所消耗的资源分配合适的事务管理器。通过自动地分配事务管理器，WCF将事务管理从服务代码和用到的事务协议中解耦出来，开发者不必为事务管理器而苦恼。下面分别介绍下这三种事务管理器。

- LTM：它只管理本地事务，即在一个单独应用程序域内的事务。LTM只能管理在一个单独服务内的事务，该服务不能将事务传递给其他服务。LTM在所有的事务管理器中，性能最好。
- KTM：与LTM一样，KTM管理的事务只能引入一个服务，并且该服务不得向其他服务传播事务。
- DTC：DTC可以管理跨越任意执行边界的事务，从本地跨越所有的边界，如进程、机器或站点的边界。DTC既可以使用OleTx协议，也可以使用WSAT协议。DTC与WCF紧密的集成一起，它是每个运行WCF的机器上默认可用的系统服务，DTC可以创建新的事务、跨机器传播事务，手机之一管理器的投票并通知资源管理器进行回滚或提交。

2.4 服务支持的4种事务模式

事务使用哪个事务由绑定的事务流属性（TransactionFlow.aspx)属性）、操作契约中的事务流选项（TransactionFlowOption.aspx)） 以及操作行为特性中的事务范围属性（TransactionScopeRequired.aspx)）共同决定。TransactionFlow属性有2个值，true 或false，TransactionFlowOption有三个值，NotAllowed、Allowed和Mandatory，TransactionScopeRequired有两个值，true或false。所以一共有12种（232）可能的配置设置。在这些配置设置中，有4种不满足要求的，例如在绑定中设置TransactionFlow属性为false，却设置TransactionFlowOption为Mandatory。下图列出了剩下的8种情况：

绑定事务流	TransactionFlowOption	TransactionScopeRequired	事务模式
False	Allowed	False	None
False	Allowed	True	Service
False	**NotAllowed**	**False**	**None**
False	**NotAllowed**	**True**	**Service**
True	Allowed	False	None
True	**Allowed**	**True**	**Client/Service**
True	Mandatory	False	None
True	**Mandatory**	**True**	**Client**

上图中的8中排列组合实际最终只产生了四种事务传播模式，这4种传播模式为：Client/Service、Client、Service和None。上图黑体字指出各种模式推荐的配置设置。在设计应用程序时，每种模式都有它自己的适用场景。对于除None模式的其他三种模式的推荐配置详细介绍如下所示：

- Client/Service：最常见的一种事务模型，通常由客户端或服务本身启用一个事务。设置步骤：

(1) 选择一个支持事务的Binding，设置 TransactionFlow = true。 (2) 设置 TransactionFlow(TransactionFlowOption.Allowed)。 (3) 设置 OperationBehavior(TransactionScopeRequired=true)。

- Client：强制服务必须参与事务，而且必须是客户端启用事务。设置步骤：

(1) 选择一个支持事务的Binding，设置 TransactionFlow = true。 (2) 设置 TransactionFlow(TransactionFlowOption.Mandatory)。 (3) 设置 OperationBehavior(TransactionScopeRequired=true)。

- Service：服务必须启用一个根事务，且不参与任何外部事务。设置步骤：

(1) 选择任何一种Binding，设置 TransactionFlow = false(默认)。 (2) 设置 TransactionFlow(TransactionFlowOption.NotAllowed)。 (3) 设置 OperationBehavior(TransactionScopeRequired=true)。

三、WCF事务服务的实现

上面内容对WCF中事务进行了一个详细的介绍，下面具体通过一个实例来说明WCF中如何实现对事务的支持。首先还是按照前面博文中介绍的步骤来实现该实例。

第一步：创建WCF契约和契约的实现，具体的实现代码如下所示：

```
namespace WCFContractAndService
{
// 服务契约
    [ServiceContract(SessionMode= SessionMode.Required)]
    //[ServiceBehavior(ReleaseServiceInstanceOnTransactionComplete
    public interface IOrderService
    {
        // 操作契约
        [OperationContract]
        // 控制客户端的事务是否传播到服务
        // TransactionFlow的值会包含在服务发布的元数据上
        [TransactionFlow(TransactionFlowOption.NotAllowed)]
        List<Customer> GetCustomers();

        [OperationContract]
        [TransactionFlow(TransactionFlowOption.NotAllowed)]
        List<Product> GetProducts();

        [OperationContract]
        [TransactionFlow(TransactionFlowOption.Mandatory)]
        string PlaceOrder(Order order);

        [OperationContract]
        [TransactionFlow(TransactionFlowOption.Mandatory)]
```

```
        string AdjustInventory(int productId, int quantity);

        [OperationContract]
        [TransactionFlow(TransactionFlowOption.Mandatory)]
        string AdjustBalance(int customerId, decimal amount);
    }

    [DataContract]
    public class Customer
    {
        [DataMember]
        public int CustomerId { get; set; }

        [DataMember]
        public string CompanyName { get; set; }

        [DataMember]
        public decimal Balance { get; set; }
    }

    [DataContract]
    public class Product
    {
        [DataMember]
        public int ProductId { get; set; }

        [DataMember]
        public string ProductName { get; set; }

        [DataMember]
        public decimal Price { get; set; }

        [DataMember]
        public int OnHand { get; set; }
    }

    [DataContract]
    public class Order
    {
        [DataMember]
        public int CustomerId { get; set; }

        [DataMember]
        public int ProductId { get; set; }

        [DataMember]
        public decimal Price { get; set; }

        [DataMember]
        public int Quantity { get; set; }

        [DataMember]
        public decimal Amount { get; set; }
```

```
    }
}

namespace WCFContractAndService
{
// 服务实现
    [ServiceBehavior(
        TransactionIsolationLevel = IsolationLevel.Serializable,
        TransactionTimeout= "00:00:30",
        InstanceContextMode = InstanceContextMode.PerSession,
        TransactionAutoCompleteOnSessionClose = true)]
    public class OrderService :IOrderService
    {
        private List<Customer> customers = null;
        private List<Product> products = null;
        private int orderId = 0;
        private string conString = Properties.Settings.Default.Tran

        public List<Customer> GetCustomers()
        {

            customers = new List<Customer>();
            using (var cnn = new SqlConnection(conString))
            {
                using (var cmd = new SqlCommand("SELECT * " + "FROM
                {
                    cnn.Open();
                    using (SqlDataReader CustomersReader = cmd.Exec
                    {
                        while (CustomersReader.Read())
                        {
                            var customer = new Customer();
                            customer.CustomerId = CustomersReader.(
                            customer.CompanyName = CustomersReader.
                            customer.Balance = CustomersReader.GetD
                            customers.Add(customer);
                        }
                    }
                }
            }

            return customers;
        }

        public List<Product> GetProducts()
        {
            products = new List<Product>();
            using (var cnn = new SqlConnection(conString))
            {
                using (var cmd = new SqlCommand(
                  "SELECT * " +
                  "FROM Products ORDER BY ProductId", cnn))
                {
```

```
                    cnn.Open();
                    using (SqlDataReader productsReader =
                      cmd.ExecuteReader())
                    {
                        while (productsReader.Read())
                        {
                            var product = new Product();
                            product.ProductId = productsReader.GetI
                            product.ProductName = productsReader.Ge
                            product.Price = productsReader.GetDecin
                            product.OnHand = productsReader.GetInt1
                            products.Add(product);
                        }
                    }
                }
            }
            return products;
        }

        // 设置服务的环境事务
        // 使用Client模式,即使用客户端的事务
        [OperationBehavior(TransactionScopeRequired =true, Transact
        public string PlaceOrder(Order order)
        {
            using (var conn = new SqlConnection(conString))
            {
                var cmd = new SqlCommand(
                  "Insert Orders (CustomerId, ProductId, " +
                  "Quantity, Price, Amount) " + "Values( " +
                  "@customerId, @productId, @quantity, " +
                  "@price, @amount)", conn);

                cmd.Parameters.Add(new SqlParameter(
                  "@customerId", order.CustomerId));
                cmd.Parameters.Add(new SqlParameter(
                  "@productid", order.ProductId));
                cmd.Parameters.Add(new SqlParameter(
                  "@price", order.Price));
                cmd.Parameters.Add(new SqlParameter(
                  "@quantity", order.Quantity));
                cmd.Parameters.Add(new SqlParameter(
                  "@amount", order.Amount));

                try
                {
                    conn.Open();
                    if (cmd.ExecuteNonQuery() <= 0)
                    {
                        return "The order was not placed";
                    }

                    cmd = new SqlCommand(
                      "Select Max(OrderId) From Orders " +
```

```
                    "Where CustomerId = @customerId", conn);
                  cmd.Parameters.Add(new SqlParameter(
                    "@customerId", order.CustomerId));

                  using (SqlDataReader reader = cmd.ExecuteReader
                  {
                      while (reader.Read())
                      {
                          orderId = Convert.ToInt32(reader[0].ToS
                      }
                  }
                  return string.Format("Order {0} was placed", or
              }
              catch (Exception ex)
              {
                  throw new FaultException(ex.Message);
              }
          }
      }

      // 使用Client模式,即使用客户端的事务
      [OperationBehavior(TransactionScopeRequired = true, Transac
      public string AdjustInventory(int productId, int quantity)
      {
          using (var conn = new SqlConnection(conString))
          {
              var cmd = new SqlCommand(
                "Update Products Set OnHand = " +
                "OnHand - @quantity " +
                "Where ProductId = @productId", conn);
              cmd.Parameters.Add(new SqlParameter(
                "@quantity", quantity));
              cmd.Parameters.Add(new SqlParameter(
                "@productid", productId));

              try
              {
                  conn.Open();
                  if (cmd.ExecuteNonQuery() <= 0)
                  {
                      return "The inventory was not updated";
                  }
                  else
                  {
                      return "The inventory was updated";
                  }
              }
              catch (Exception ex)
              {
                  throw new FaultException(ex.Message);
              }
          }
      }
```

```
        // 使用Client模式,即使用客户端的事务
        [OperationBehavior(TransactionScopeRequired = true, Transac
        public string AdjustBalance(int customerId, decimal amount)
        {
            using (var conn = new SqlConnection(conString))
            {
                var cmd = new SqlCommand(
                  "Update Customers Set Balance = " +
                  "Balance - @amount " +
                  "Where CustomerId = @customerId", conn);
                cmd.Parameters.Add(new SqlParameter(
                  "@amount", amount));
                cmd.Parameters.Add(new SqlParameter(
                  "@customerId", customerId));

                try
                {
                    conn.Open();
                    if (cmd.ExecuteNonQuery() <= 0)
                    {
                        return "The balance was not updated";
                    }
                    else
                    {
                        return "The balance was updated";
                    }
                }
                catch (Exception ex)
                {
                    throw new FaultException(ex.Message);
                }
            }
        }
    }
}
```

上面的服务契约和服务实现与传统的实现没什么区别。这里使用IIS来宿主WCF服务。

第二步：宿主的实现。创建一个空的Web的项目，并添加WCF服务文件，具体内容如下所示：

对应的Web.config的内容如下所示：

```
<configuration>
    <system.web>
      <compilation debug="true" targetFramework="4.5" />
      <httpRuntime targetFramework="4.5" />
    </system.web>

    <system.serviceModel>
        <behaviors>
            <serviceBehaviors>
                <behavior name="OrderServiceBehavior">

                    <serviceMetadata httpGetEnabled="true" httpsGet
                    <serviceDebug includeExceptionDetailInFaults="t
                </behavior>
            </serviceBehaviors>
        </behaviors>
      <bindings>
        <wsHttpBinding>
          <!--通过设置transactionFlow属性为true来使绑定支持事务传播；对于
          <binding name="wsHttpBinding" transactionFlow="true">
            <!--启用消息可靠性选项-->
            <!--<reliableSession enabled="true"/>-->
          </binding>

        </wsHttpBinding>

      </bindings>
      <services>
        <service name="WCFContractAndService.OrderService" behavior
          <endpoint address="" binding="wsHttpBinding" bindingConfi
        </service>
      </services>
        <serviceHostingEnvironment aspNetCompatibilityEnabled="true
            multipleSiteBindingsEnabled="true" />
    </system.serviceModel>
</configuration>
```

这里采用了wsHttpBinding绑定，并设置其transactionFlow属性为true使其支持事务传播。接下来看看客户端的实现。

第三步：WCF客户端的实现，通过添加服务引用的方式来生成代理类。这里的客户端是WinForm程序。

```
1 public partial class Form1 : Form
2     {
3         public Form1()
4         {
5             InitializeComponent();
6         }
7
```

```
 8         private Customer customer = null;
 9         private List<Customer> customers = null;
10         private Product product = null;
11         private List<Product> products = null;
12         private OrderServiceClient proxy = null;
13         private Order order = null;
14         private string result = String.Empty;
15 
16         private void Form1_Load(object sender, EventArgs e)
17         {
18             proxy = new OrderServiceClient("WSHttpBinding_IOrder
19             GetCustomersAndProducts();
20         }
21 
22         private void GetCustomersAndProducts()
23         {
24             customers = proxy.GetCustomers().ToList<Customer>();
25             customerBindingSource.DataSource = customers;
26 
27             products = proxy.GetProducts().ToList<Product>();
28             productBindingSource.DataSource = products;
29         }
30 
31         private void placeOrderButton_Click(object sender, Event
32         {
33             customer = (Customer)this.customerBindingSource.Curr
34             product = (Product)this.productBindingSource.Current
35             Int32 quantity = Convert.ToInt32(quantityTextBox.Tex
36 
37             order = new Order();
38             order.CustomerId = customer.CustomerId;
39             order.ProductId = product.ProductId;
40             order.Price = product.Price;
41             order.Quantity = quantity;
42             order.Amount = order.Price * Convert.ToDecimal(order
43 
44             // 事务处理
45             using (var tranScope = new TransactionScope())
46             {
47                 proxy = new OrderServiceClient("WSHttpBinding_IO
48                 {
49                     try
50                     {
51                         result = proxy.PlaceOrder(order);
52                         MessageBox.Show(result);
53 
54                         result = proxy.AdjustInventory(product.P
55                         MessageBox.Show(result);
56 
57                         result = proxy.AdjustBalance(customer.Cu
58                           Convert.ToDecimal(quantity) * order.Pr
59                         MessageBox.Show(result);
60 
```

```
61                          proxy.Close();
62                          tranScope.Complete(); // Cmmmit transact
63                      }
64                      catch (FaultException faultEx)
65                      {
66                          MessageBox.Show(faultEx.Message +
67                            "\n\nThe order was not placed");
68
69                      }
70                      catch (ProtocolException protocolEx)
71                      {
72                          MessageBox.Show(protocolEx.Message +
73                            "\n\nThe order was not placed");
74                      }
75                  }
76              }
77
78              // 成功提交后强制刷新界面
79              quantityTextBox.Clear();
80              try
81              {
82                  proxy = new OrderServiceClient("WSHttpBinding_IO
83                  GetCustomersAndProducts();
84              }
85              catch (FaultException faultEx)
86              {
87                  MessageBox.Show(faultEx.Message);
88              }
89          }
90      }
```

从上面代码可以看出，WCF事务的实现是利用TransactionScope.aspx)事务类来完成的。下面让我们看看程序的运行结果。在运行程序之前，我们必须运行SQL脚本来创建程序中的使用的数据库，具体的脚本如下所示：

```
 1 USE [TransactionsDemo]
 2 GO
 3 /****** Object:  Table [dbo].[Customers]    Script Date: 01/15/2
 4 SET ANSI_NULLS ON
 5 GO
 6 SET QUOTED_IDENTIFIER ON
 7 GO
 8 CREATE TABLE [dbo].[Customers](
 9     [CustomerId] [int] IDENTITY(1,1) NOT NULL,
10     [Name] [nvarchar](20) NOT NULL,
11     [Balance] [smallmoney] NOT NULL, check(Balance >= 0),
12  CONSTRAINT [PK_Customers] PRIMARY KEY CLUSTERED
13 (
14     [CustomerId] ASC
15 )WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF, IGNORE_D
```

```
16 ) ON [PRIMARY]
17 GO
18 /****** Object:  Table [dbo].[Products]    Script Date: 01/15/20
19 SET ANSI_NULLS ON
20 GO
21 SET QUOTED_IDENTIFIER ON
22 GO
23 CREATE TABLE [dbo].[Products](
24     [ProductId] [int] IDENTITY(1,1) NOT NULL,
25     [Name] [nvarchar](20) NOT NULL,
26     [Price] [smallmoney] NOT NULL,
27     [OnHand] [smallint] NOT NULL, check(OnHand >= 0),
28  CONSTRAINT [PK_Products] PRIMARY KEY CLUSTERED
29 (
30     [ProductId] ASC
31 )WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF, IGNORE_D
32 ) ON [PRIMARY]
33 GO
34 /****** Object:  Table [dbo].[Orders]    Script Date: 01/15/2009
35 SET ANSI_NULLS ON
36 GO
37 SET QUOTED_IDENTIFIER ON
38 GO
39 CREATE TABLE [dbo].[Orders](
40     [OrderId] [int] IDENTITY(1,1) NOT NULL,
41     [CustomerId] [int] NOT NULL,
42     [ProductId] [int] NOT NULL,
43     [Quantity] [smallint] NOT NULL,
44     [Price] [smallmoney] NOT NULL,
45     [Amount] [smallmoney] NOT NULL,
46  CONSTRAINT [PK_Orders] PRIMARY KEY CLUSTERED
47 (
48     [OrderId] ASC
49 )WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF, IGNORE_D
50 ) ON [PRIMARY]
51 GO
52 INSERT Customers (Name, Balance) VALUES ('Contoso', 10000)
53 INSERT Customers (Name, Balance) VALUES ('Northwind', 25000)
54 INSERT Customers (Name, Balance) VALUES ('Litware', 50000)
55 INSERT Products (Name, Price, OnHand) VALUES ('Wood', 100, 1000)
56 INSERT Products (Name, Price, OnHand) VALUES ('Wallboard', 200,
57 INSERT Products (Name, Price, OnHand) VALUES ('Pipe', 500, 5000)
58 GO
```

生成程序使用的数据库之后，按F5运行WCF客户端程序，并在出现的界面中购买Wood材料100，运行结果如下图所示：

Form1

Select a customer

Customer	Balance
Contoso	10000.0000
Northwind	25000.0000
Litware	50000.0000

Select a product

Product	Price	OnHand
Wood	100.0000	1000
Wallboard	200.0000	2500
Pipe	500.0000	5000

Quantity 100 Place order

单击Place order按钮后，即执行下订单操作，如果订单成功后，将会更新产品的库存和用户的余额，你将看到如下图所示的运行结果：

Form1

Select a customer

Customer	Balance
Contoso	0
Northwind	25000.0000
Litware	50000.0000

Select a product

Product	Price	OnHand
Wood	100.0000	900
Wallboard	200.0000	2500
Pipe	500.0000	5000

Quantity

Place order

四、小结

到这里，关于WCF中事务的介绍就结束了。WCF支持四种事务模式，Client/Service、Client、Service和None，对于每种模式都有其不同的配置。一般尽量使用Client/Service或Client事务模式。WCF事务的实现借助于已有的System.Transaction实现本地事务的编程，而分布式事务则借助MSDTC分布式事务协调机制来实现。WCF提供了支持事务传播的绑定协议包括：wsHttpBinding、WSDualHttpBinding、WSFederationBinding、NetTcpBinding和NetNamedPipeBinding，最后两个绑定允许选择WS-AT协议或OleTx协议，而其他绑定都使用标准的WS-AT协议。在一一篇博文将分享WCF对消息队列的支持。

本文所有源代码：WCFTransaction.zip

跟我一起学WCF(11)——WCF中队列服务详解

一、引言

在前面的WCF服务中，它都要求服务与客户端两端都必须启动并且运行，从而实现彼此间的交互。然而，还有相当多的情况希望一个面向服务的应用中拥有离线交互的能力。WCF通过服务队列的方法来支持客户端和服务之间的离线工作，客户端将消息发送到一个队列中，再由服务对它们进行处理。下面让我们具体看看WCF中的队列服务。

二、WCF队列服务的优势

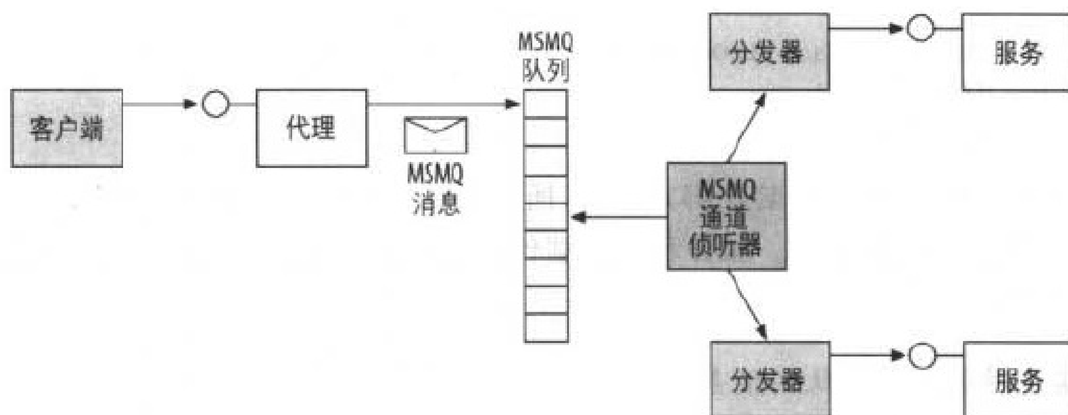
在介绍WCF队列服务之前，首先需要了解微软消息队列（MSMQ）。MSMQ是在多个不同应用之间实现相互通信的一种异步传输模式，相互通信的应用可以分布在同一台机器，也可以分布在相连的网络环境。它的实现原理是：客户端将消息发送到一个容器中，然后把它保存到一个系统公用空间的消息队列（Message Queue）中，本地或异地的服务再从该队列中取出发送给它的消息进行处理。更多详细内容可以参考我的博文：跟我一起学WCF(1)——MSMQ消息队列。

WCF框架对MSMQ进行了集成和扩展，MSMQ支持离线消息模式，并且在WCF框架下，提供了基于http桥的internet网络队列服务的调用扩展。从而赋予了WCF队列服务以下几点优势：

1. 支持离线消息模式。因为WCF框架集成了MSMQ，所以WCF队列服务自然也支持离线消息。
2. 支持将操作分解。WCF支持将工作分解为多个操作放入队列中，可改善系统的可用性和吞吐量。
3. 提供对失败的事务做善后处理。当我们的业务事务需要几个小时或几天完成的时候，我们通常将它分为至少2个事务。第一个事务将需要立即完成的工作放入队列，而第二个事务用于验证第一个事务是否成功，并在必要的情况下对失败的事务进行善后处理。
4. 支持负载平衡。可以把过载的客户端请求放入队列，空闲的时候进行处理，这样可以平衡系统的吞吐量，改善性能。

三、WCF队列服务通信框架

WCF使用NetMsmqBinding来支持消息队列通信。当客户端调用服务时，客户端消息会被封装为MSMQ消息，发送到系统公用的消息队列中，服务宿主在运行状态下会启动通道监听器来检测消息队列消息，如果发现对应的消息，则会从队列里取出消息，使用分发器转发给对应的服务，具体的通信框架如下图所示：



如果宿主离线，消息会被放入队列，等待下一次宿主联机时，在执行消息分发给指定WCF服务处理。

另外WCF还提供了MsmqIntegrationBinding.aspx)类，该类用于需要将WCF 应用和现有的基于MSMQ的应用集成的情况。WCF应用可利用该绑定向现有的MSMQ应用程序发生消息，或从这些应用程序接收消息。

四、利用WCF队列服务来实现离线操作

前面介绍WCF队列服务的优势和它的通信框架，下面具体通过一个例子来诠释WCF队列服务的实现。我们还是按照前面文章介绍的三个步骤来实现该实例。

第一步：定义契约和实现服务。具体的实现代码如下所示：

```
1 [ServiceContract]
2     public interface IWCFMSMQService
3     {
4         // 操作契约，必须为单向操作
5         [OperationContract(IsOneWay = true)]
6         void SayHello(string message);
7     }
8
9 // 契约实现
10     public class WCFMSMQService : IWCFMSMQService
11     {
12         public WCFMSMQService()
13         {
14             Console.WriteLine("WCF MSMQ Service instance was cre
15         }
16
17         #region IOrderProcessor Members
18
19         [OperationBehavior(TransactionScopeRequired = true, Tran
20         public void SayHello(string message)
21         {
22             Console.WriteLine("Hello! {0},调用WCF操作的时间为：{1}"
23         }
24
25         #endregion
26     }
```

上面代码需要注意一点：WCF操作必须定义为单向操作，因为要实现的是一个队列服务，其特点为异步、离线，无返回值。所以要设置IsOneWay属性为true。

第二步：实现宿主。这里仍然使用控制台应用程序作为WCF队列服务的宿主，具体的实现代码如下所示：

```
1 namespace WCFConsoleHost
2 {
3     class Program
4     {
5         static void Main(string[] args)
6         {
7             string path = @".\private$\LearningHardWCFMSMQ";
8             if (!MessageQueue.Exists(path))
9             {
10                MessageQueue.Create(path, true);
11            }
12
13            using (ServiceHost host = new ServiceHost(typeof(WCF
14            {
15                host.Opened += delegate
16                {
17                    Console.WriteLine("Service has begun to list
18                };
19
20                host.Open();
21
22                Console.Read();
23            }
24        }
25    }
26 }
```

对应的配置信息如下所示：

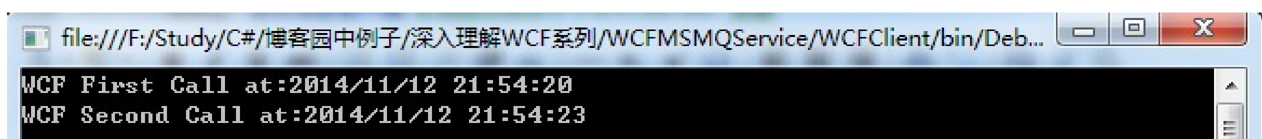
```
<configuration>
  <system.serviceModel>
    <behaviors>
      <serviceBehaviors>
        <behavior name="msmqServiceBehavior">
          <serviceMetadata httpGetEnabled="true" />
        </behavior>
      </serviceBehaviors>
    </behaviors>
    <bindings>
      <netMsmqBinding>
        <binding name="msmqBinding">
          <security>
            <transport msmqAuthenticationMode="None" msmqProtectior
            <message clientCredentialType="None"/>
          </security>
        </binding>
      </netMsmqBinding>
    </bindings>
    <services>
      <service behaviorConfiguration="msmqServiceBehavior" name="WC
        <endpoint address="net.msmq://localhost/private/LearningHar
          bindingConfiguration="msmqBinding" contract="WCFContractA
        <!--发布服务元数据的终结点-->
        <endpoint address="mex" binding="mexHttpBinding" contract='
        <host>
          <baseAddresses>
            <add baseAddress="http://localhost:9003/" />
          </baseAddresses>
        </host>
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

第三步：WCF客户端的实现。首先以管理员权限启动宿主，然后通过添加服务引用的方式来生成代理客户端类，具体在添加服务引用窗口地址栏输入：http://localhost:9003/mex来添加服务引用，添加服务引用成功后将生成代理类，通过代理类来对WCF服务进行访问，具体的实现代码如下所示：

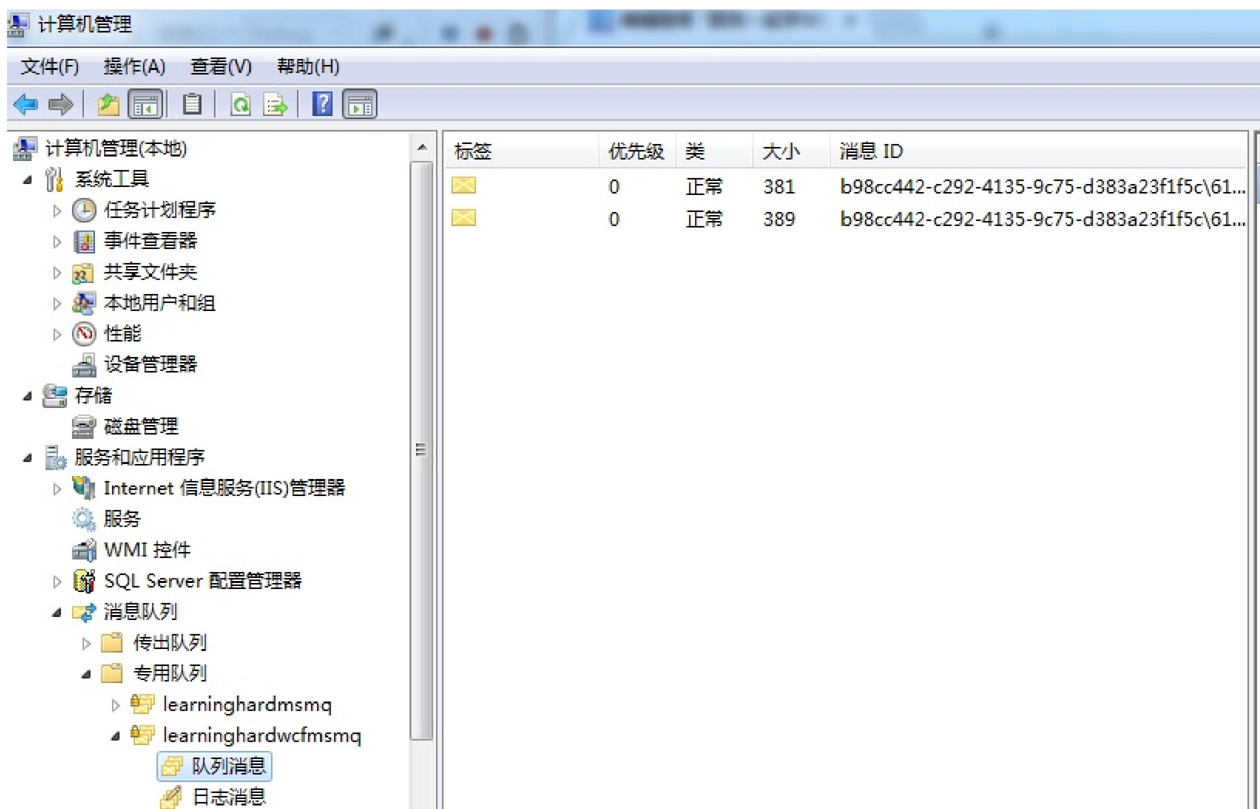
```
 1 namespace WCFClient
 2 {
 3     class Program
 4     {
 5         static void Main(string[] args)
 6         {
 7             WCFMSMQServiceClient proxy = new WCFMSMQServiceClier
 8             using (TransactionScope scope = new TransactionScope
 9             {
10                 Console.WriteLine("WCF First Call at:{0}", DateT
11                 proxy.SayHello("World");
12 
13                 Thread.Sleep(2000);//客户端休眠两秒，继续下一次调用
14                 Console.WriteLine("WCF Second Call at:{0}", Date
15                 proxy.SayHello("Learning Hard");
16 
17                 scope.Complete();
18             }
19 
20             Console.Read();
21         }
22     }
23 }
```

经过上面三步，我们就完成了一个WCF队列服务程序。下面让我们看看该程序的运行结果。

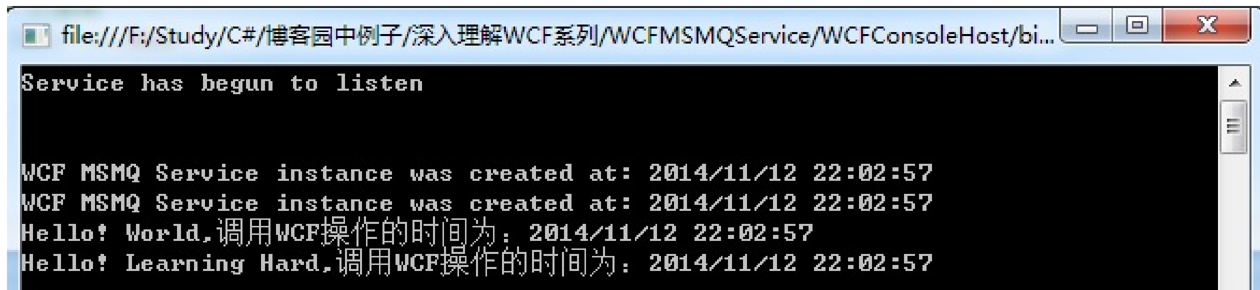
因为WCF队列服务是对MSMQ的集成和扩展，所以此时该程序客户端可以在WCF服务离线的情况也可运行，即WCF服务宿主不启动，客户端也可以正常运行，这点与前面介绍的WCF程序完成不同，因为前面的WCF程序，如果宿主程序不启动而直接启动客户端程序，则客户端程序运行时将会发生异常。该程序之所以不会发生异常的原因在于，此时客户端程序是直接与消息队列进行交互的，而不是直接与WCF服务进行交互。此时只运行WCF客户端，你将看到如下图所示的运行结果：



同时，你在消息队列的专有队列的队列消息中将看到两条未处理的消息信息，具体效果如下图所示：



因为WCF服务宿主还没有启动，所以客户端发送的消息信息还没有被取出处理，接下来，我们启动下服务宿主来对队列中的消息进行处理，此时你可以选择关闭客户端（当然你也可以不关闭）。具体的WCF服务宿主的运行结果如下图所示：



此时，如果刷新消息队列的专有队列中的队列消息，你将看不到任何消息了，这是因为WCF服务从消息队列取出消息进行处理了，即消息完成了出队的操作。

五、总结

到这里，WCF队列服务的介绍也就结束了。本文主要介绍了WCF集成了MSMQ的功能，WCF可以利用netMsmqBinding来实现离线服务。其实现程序与MSMQ程序实现差不多，关于MSMQ的相关内容也可以参考我的另一篇博文：跟我一起学WCF(1)——MSMQ消息队列。在下一篇博文中将分享WCF对Rest服务的支持和实现。

本文所有源码：WCFMSMQService.zip

跟我一起学WCF(12)——WCF中Rest服务入门

一、引言

要将Rest与.NET Framework 3.0配合使用，还需要构建基础架构的一些部件。在.NET Framework 3.5中，WCF在System.ServiceModel.Web组件中新增了编程模型和这些基础架构部件。

新编程模型有两个主要的新属性：WebGetAttribute.aspx)和WebInvokeAttribute.aspx)，还有一个URI模板机制，帮助你声明每种方法响应使用的URI和动词。.NET Framework还提供了一个新的绑定（WebHttpBinding）和新的行为（WebHttpBehavior），此外，还提供了WebServiceHost和WebServiceHostFactory类来对Rest服务进行支持。下面让我们具体看看WCF目前对Rest服务的支持和实现。

二、REST服务是什么

对于这个问题，百度下有很多答案，这里给出百度百科中一个详细介绍的链接：Rest服务。我的理解的Rest服务是：以前我们都是把WCF服务抽象为操作的概念，而Rest最早是由Roy Thomas Fielding 在他的博士论文（“体系结构风格和基于网络软件体系的设计”）中提出的。Rest服务是将服务抽象为资源，每个资源都有一个唯一的统一资源标识符（URI），我们不再是通过调用操作的方式与服务进行交互了，而是通过HTTP标准动词（GET、POST、PUT和DELETE）的统一接口来完成。总之一句话概括，Rest服务换了一种思维方式，把服务也当成一种资源，通过Get、Post、Put和Delete这些HTTP动词来进行交互。这样，问题就来了，Rest服务具有什么好处呢？即我们为什么要去关注Rest和实现它呢？

Rest优势就在于其使用极其简单，Rest服务要求很少的编码工作量，可以减少很多不必要的工作。Rest服务主要有以下优点：

- 无需引入SOAP消息传输层，轻量级和高效率的HTTP格式可直接被应用。
- 可以轻易地在任何编程语言中实现，尤其是在JS中。使用SOAP的服务与JS交互非常繁琐，而使用Rest服务与JS交互非常简单。
- 可以不使用任何编程语言就能访问服务，而只需要使用Web浏览器即可。
- 更好的性能和缓存支持。使用Rest服务可以改善响应时间和改善用户体验。
- 可扩展性和无状态性。Rest服务基于HTTP协议，每个请求都是独立的，一旦被调用，服务器不保留任何会话，这样可以更具响应性，通过减少通讯状态的维护工作来提供服务的可扩展性。

三、WXF和Asp.net Web API的比较

WCF是微软为生成面向服务的应用程序而提供的统一编程模型。Asp.net Web API是一个用来方便生成HTTP服务的框架，这些服务可供广泛的客户端访问，包括浏览器和移动设备。Asp.net Web API用于在.NET平台上生成Restful应用程序的理想平台。到这里问题又来了，Rest服务与SOAP服务的区别又是什么呢？

Rest相对于SOAP服务使用更加简单，对于开发者来说，学习成本较低，而SOAP作为一种古老的Web服务技术，近期内还不回退出历史舞台，而且随着SOAP 1.2的出现，SOAP 1.1中的一些缺点已得到改进。

REST目前只基于HTTP和HTTPS协议，而SOAP可支持任何传输协议，包括HTTP/HTTPS、TCP、SMTP等协议。另外Rest服务除了能使用XML作为数据承载外，还有JSON，RSS和ATOM形式。

Rest与SOAP对应的比较如下所示：

1. SOAP是一种工业标准，面对的应用需求时RPC（远程过程调用），而Rest面对的应用需求是分布式Web系统。
2. Rest服务强调数据，请求和响应消息都是数据的封装，而SOAP服务更强调接口，SOAP消息封装的是过程调用。Rest是面向资源的，而SOAP是面向接口的。
3. Rest架构下，HTTP是承载协议，也是应用协议，而SOAP架构下，HTTP只是承载协议，SOAP才是应用协议。

那在什么情况下使用Rest，什么情况下使用SOAP呢？这要看具体的实际情况。具体应用场景如下所示：

- 远程调用用SOAP。如果服务是作为一种功能提供，客户端调用服务是为了执行一个功能，用SOAP，比如常见的需求是认证授权。而数据服务用Rest。
- 要更多的考虑非功能需求时使用SOAP，如需考虑安全、传输和协议等需求的情况下。
- 低带宽、客户端的处理能力受限的场合可以考虑使用Rest。如在PDA或手机上消费服务。

介绍了Rest与SOAP的区别之后，让我们回到WCF与Asp.net Web API的比较上来，具体两者功能之间的对比如下图所示：

WCF	ASP.NET Web API
启用支持多种传输协议（HTTP、TCP、UDP 和自定义传输）的生成服务，并允许在这些服务之间切换。	仅限 HTTP。 用于 HTTP 的第一类编程模型。更适合从各种浏览器、移动设备等进行访问，可实现大范围访问
启用支持同一消息类型的多种编码（文本、MTOM 和二进制）的生成服务，并允许在这些服务之间切换。	Uses basic protocol and formats such as HTTP, WebSockets, SSL, JQuery, JSON, and XML. 建议更好的翻译
支持采用 WS-* 标准的生成服务，如可靠消息传递、事务、消息安全性。	使用基本协议和格式，如 HTTP、WebSocket、SSL、JQuery、JSON 和 XML。 不支持较高级别的协议，如消息传递或事务。
支持请求-答复、单向和双工消息交换模式。	HTTP 是请求/响应，不过，通过 SignalR 和 WebSocket 集成，可集成其他模式。
可以在 WSDL 中描述 WCF SOAP 服务，从而可通过自动工具，针对具有复杂架构的服务来生成客户端代理。	可通过各种方法来描述 Web API：从用于描述代码片段的自动生成的 HTML 帮助页，直至用于 OData 集成 API 的结构化元数据。
随 .NET Framework 一起提供。	随 .NET Framework 一起提供，但它是一个开放源代码程序，也可通过独立下载以带外方式获得。

使用 WCF 可创建可靠、安全的 Web 服务，这些服务可通过各种传输方式来访问。 使用 ASP.NET Web API 可创建基于 HTTP 的服务，这些服务可从各种客户端来访问。 如果要创建和设计新的 REST 样式服务，请使用 ASP.NET Web API。 虽然 WCF 针对编写 REST 样式服务提供了一些支持，但 ASP.NET Web API 中的 REST 支持更加完整，并且，所有将来的 REST 功能改进都将在 ASP.NET Web API 中进行。

四、WCF中实现Rest服务

WCF 3.5中对Rest服务也做了支持，主要提供了WebHttpBinding.aspx)来对Rest进行支持，下面我们通过一个具体的实例来看看如何在WCF中实现一个Rest服务。我们还是按照之前三个步骤来创建该实例。

第一步：创建Rest服务接口和实现。具体的实现代码如下所示。

服务契约代码如下所示：

```
1  [ServiceContract(Namespace ="http://www.cnblogs.com/zhili/")]
2     public interface IEmployees
3     {
4         // 契约操作不再使用操作契约的方式来标识，而是使用WebGetAttribut
5         [WebGet(UriTemplate = "all")]
6         IEnumerable<Employee> GetAll();
7
8         [WebGet(UriTemplate = "{id}")]
9         Employee Get(string id);
10
11        [WebInvoke(UriTemplate="/", Method="PUT")]
12        void Create(Employee employee);
13
14        [WebInvoke(UriTemplate = "/", Method = "POST")]
15        void Update(Employee employee);
16
17        [WebInvoke(UriTemplate = "/", Method = "DELETE")]
18        void Delete(string id);
19    }
20
21    [DataContract(Namespace = "http://www.cnblogs.com.zhili/")]
22    public class Employee
23    {
24        [DataMember]
25        public string Id { get; set; }
26
27        [DataMember]
28        public string Name { get; set; }
29
30        [DataMember]
31        public string Department { get; set; }
32
33        [DataMember]
34        public string Grade { get; set; }
35
36        public override string ToString()
37        {
38            return string.Format("ID: {0,-5}姓名：{1,-5}部门：{2,
39        }
40    }
```

从上面代码可以看出，Rest服务不再使用OperactionContract的方式来标识操作了，此时在Rest架构下，每个操作都被当做一种资源对待，所以这里的操作使用了WebGetAttribute特性和WebInvokeAttribute来标识。下面具体看看契约的具体实现，即Rest服务的实现代码。

```
 1 namespace WCFContractAndService
 2 {
 3     public class EmployeesService : IEmployees
 4     {
 5         private static IList<Employee> employees = new List<Empl
 6         {
 7             new Employee{ Id = "0001", Name = "LearningHard", De
 8             new Employee{Id = "0002", Name = "张三", Department
 9         };
10 
11         public Employee Get(string id)
12         {
13             Employee employee = employees.FirstOrDefault(e => e.
14             if (null == employee)
15             {
16                 WebOperationContext.Current.OutgoingResponse.Sta
17             }
18             return employee;
19         }
20 
21         public IEnumerable<Employee> GetAll()
22         {
23             return employees;
24         }
25 
26         public void Create(Employee employee)
27         {
28             employees.Add(employee);
29         }
30 
31 
32         public void Update(Employee emp)
33         {
34             this.Delete(emp.Id);
35             employees.Add(emp);
36         }
37 
38         public void Delete(string id)
39         {
40             Employee employee = this.Get(id);
41             if (null != employee)
42             {
43                 employees.Remove(employee);
44             }
45         }
46     }
47 }
```

第二步：实现Rest服务宿主。这里还是使用控制台程序来作为宿主程序，主要的实现代码如下所示：

```
namespace RestServiceHost
{
    class Program
    {
        static void Main(string[] args)
        {
            // Rest服务使用WebServiceHost类来为服务提供宿主
            using (WebServiceHost webHost = new WebServiceHost(type
            {
                webHost.Opened += delegate
                {
                    Console.WriteLine("Rest Employee Service 开启成功
                };

                webHost.Open();
                Console.Read();
            }
        }
    }
}
```

对应的配置文件内容如下所示：

```
<configuration>
  <system.serviceModel>
    <services>
      <service name="WCFContractAndService.EmployeesService">
        <!--这里Rest服务只能使用WebHttpBinding来作为绑定-->
        <endpoint address="http://localhost:9003/employeeService"
                  binding="webHttpBinding" contract="RestContract.I
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

第三步：实现Rest服务调用客户端。这里通过通道工厂的方式来创建代理对象的，具体的实现代码如下所示：

```
1 namespace WCFClient
2 {
3     class Program
4     {
5         static void Main(string[] args)
6         {
7             using (ChannelFactory<IEmployees> channelFactory = n
8             {
9                 // 创建代理类
10                IEmployees proxy = channelFactory.CreateChannel(
11
12                Console.WriteLine("所有员工信息：");
13
14                // 通过代理类来对Rest服务进行操作
15                Array.ForEach<Employee>(proxy.GetAll().ToArray()
16
17                Console.WriteLine("\n添加一个新员工{0003}:");
18                proxy.Create(new Employee
19                {
20                    Id = "0003", Name="李四", Department="财务部"
21                });
22
23                Array.ForEach<Employee>(proxy.GetAll().ToArray()
24
25                Console.WriteLine("\n修改员工（0003）信息:");
26                proxy.Update(new Employee
27                {
28                    Id = "0003", Name="李四", Department = "销售部
29                });
30                Array.ForEach<Employee>(proxy.GetAll().ToArray()
31                Console.WriteLine("\n删除员工(0002)信息：");
32                proxy.Delete("0002");
33                Array.ForEach<Employee>(proxy.GetAll().ToArray()
34
35                Console.Read();
36            }
37        }
38    }
39 }
```

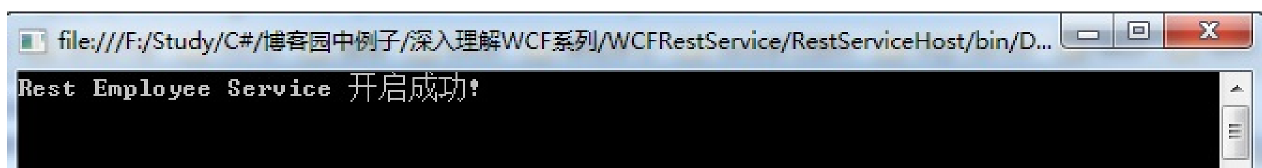
客户端对应的配置文件内容如下所示：

```
<configuration>
  <system.serviceModel>
    <behaviors>
      <endpointBehaviors>
        <behavior name="webBehavior">
         ** <webHttp/>**
        </behavior>
      </endpointBehaviors>
    </behaviors>
    <client>
      <endpoint name="employeeService" address="http://localhost:9(

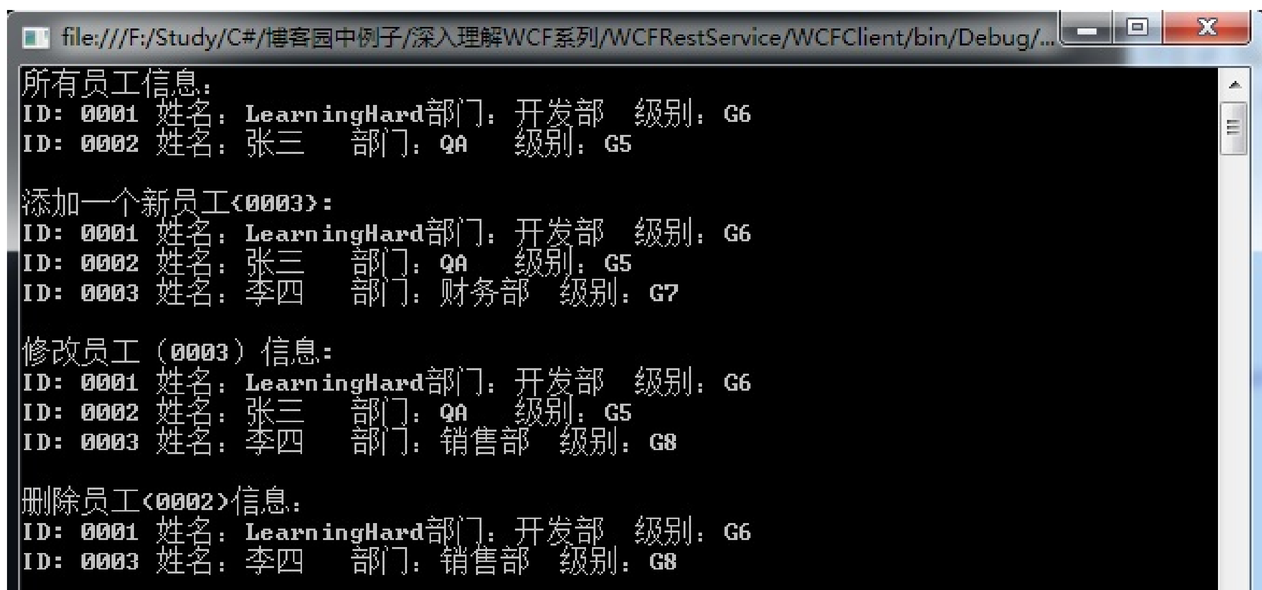
      </endpoint>
    </client>
  </system.serviceModel>
</configuration>
```

经过上面的三步，Rest服务的构建工作就完成了，下面看看该程序的运行结果。

首先以管理员权限运行服务宿主程序，运行成功后的结果如下图所示：



然后再运行客户端程序，运行成功后的结果如下图所示：



五、 总结

到这里，本文要分享的内容就结束了，同时这也是WCF系列的最后一篇博文。WCF主要通过提供几个新的API来对Rest服务的实现，这里包括WebHttpBinding类、WebGetAttribute、WebInvokeAttribute特性和WebServiceHost类等。接下来一篇博文将对本系列做一个总结。

本文所有源码：WCFRestService.zip

跟我一起学WCF(13)——WCF系列总结

引言

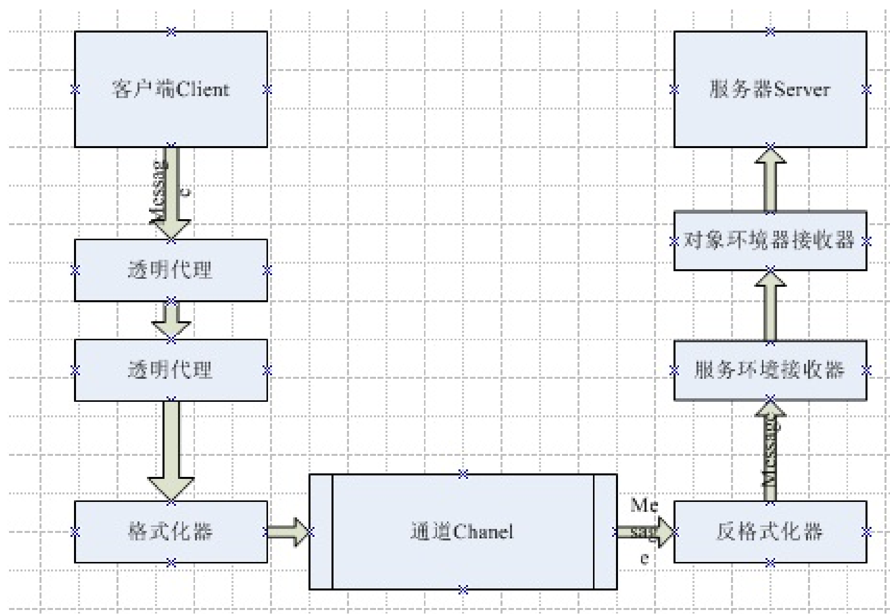
WCF是微软为了实现SOA的框架，它是对微乳之前多种分布式技术的继承和扩展，这些技术包括Enterprise Service、.NET Remoting、XML Web Service、MSMQ等。WCF推出的原因在于：微软想将不同的分布式技术整合起来，提供一个统一的编程模型，这样对于开发者来说绝对是好事。在过去的2个月时间内，我陆续写了WCF系列文章，这些文章只是自己这段时间学习WCF内容的一个学习过程和笔记，希望通过这种写博文的方式记录下来和总结。本系列并没有对WCF机制做一个深入解析，只是讲解了WCF支持的功能和实现，关于更深入的了解，我相信，只有在项目中使用遇到问题和解决问题的方式才能更深入地理解，这系列文章只是想大家对WCF有一个全面的认识。下面是本系列文章的一个索引，希望可以帮助大家进行收藏，同时也帮助我自己索引。

[第1篇] 跟我一起学WCF(1)——MSMQ消息队列

MSMQ，Microsoft Message Queue——微软消息队列，它是微软之前实现分布式技术之一。其工作原理是：客户端将消息发送到一个消息队列中，服务从该消息队列中取出消息进行处理。通过消息队列的方式，把客户端和服务之间的耦合进行隔离，最明显的好处是异步和可离线功能，缺点是：由于客户端不直接把消息发送到服务进行处理，而是把消息发送到消息队列中，从而不适合客户端需要服务实时交互的情况下，大量请求的时候，响应可能延迟。

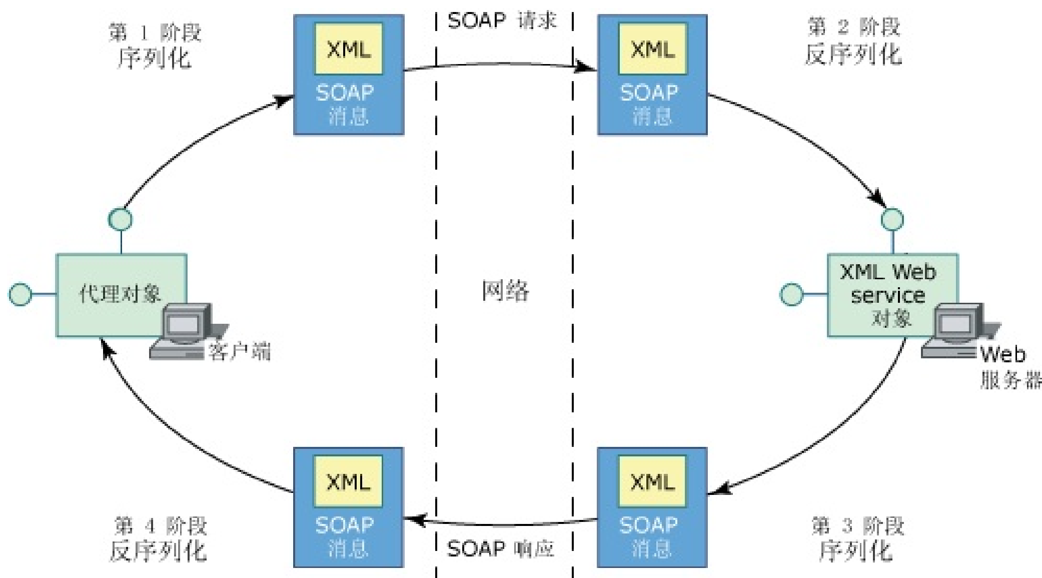
[第2篇] 跟我一起学WCF(2)——利用.NET Remoting技术开发分布式应用

.NET Remoting是微软另一种分布式技术，WCF内部实现借鉴了该技术的实现。.NET Remoting优点可以实现跨应用程序域进行通信，缺点是不支持离线功能，并只适合.NET 平台的程序进行通信。其工作原理如下图所示：



[第3篇] 跟我一起学WCF(3)——利用Web Services开发分布式应用

XML Web Service是微软另外一种分布式技术，该技术具有的优点是跨平台，跨防火墙和自我描述，像MSMQ和.NET Remoting不能跨平台，因为其传输是二进制格式的数据，而XML Web Service传输的是基于XML的文本文件。其缺点是效率地和安全性，不适合做局域网内应用。所以，一般地说，局域网可以使用MSMQ和.NET Remoting技术，而基于Internet的应用使用XML Web Service。其实现原理如下图所示：



[第四篇] 跟我一起学WCF(4)——第一个WCF程序

之前说过，WCF是对MSMQ、.NET Remoting、XML Web Service等技术的继承和扩展，所以利用WCF既可以做基于局域网的应用，也可以做基于互联网的分布式应用。WCF最重要的概念就是终结点，服务的提供者将服务通过一个或多个终结点进行发布给服务消费者。而终结点又由地址、绑定和契约组成。

这三个要素在WCF通信中起到的作用分别是：

- 地址（**Address**）：地址标识了服务的位置，提供寻址的辅助信息和标识了服务的真实身份。Address解决了**Where the WCF service?**的问题。
- 绑定（**Binding**）：绑定实现了通信的所有细节，包括网络传输，消息编码，以及其他为实现某种功能对消息进行的相应处理，例如安全、可靠传输和事务等功能。 WCF中具有一系列的系统已定义的绑定，如BasicHttpBinding、WsHttpBinding、NetTcpBinding等。Binding解决了How to Communicate with Service？的问题。
- 契约（**Contract**）：契约是对服务操作的抽象，也是对消息交互模式以及消息结构的定义。WCF的契约大体可以分为两类，一类是对服务操作的描述；另一类是对数据的描述。服务契约(Service Contract)则属于对服务操作的描述，而后一类包括其余3中契约：数据契约（Data Contract）、消息契约（Message Contract）和错误契约（Fault Contract）。Contract解决了What function does the Service Provide?的问题。

后面的WCF文章都是对于这三个元素的扩展介绍。

[第五篇] 跟我一起学WCF(5)——深入解析服务契约[上篇]

定义WCF服务，自然第一步就是需要定义服务契约，该博文主要介绍了WCF如何实现操作重载的。其主要实现逻辑是为相同的方法定义别名，使其生成的WSDL中operation标签不同。

[第六篇] 跟我一起学WCF(6)——深入解析服务契约[下篇]

WCF如果服务中定义了契约的继承关系，通过客户端生成的代理类不会生成具有继承关系的契约结构，解决这个问题的思路就是自定义代理类，使其具有和服务契约中一样的继承结构。

[第七篇] 跟我一起学WCF(7)——WCF数据契约与序列化详解

数据契约是定义服务和客户端之间要传送的自定义类型，对于一些基本类型如String、int等内置类型都是可序列化的，所以WCF默认对这些类型可进行序列化并进行传输，但对于自定义类、结构体等类型，因为这些类型默认不支持序列化，WCF中通过DataMemberAttribute特性是自定义类型可以进行序列化传输，并在服务中能进行反序列化为对象来进行数据的处理。WCF中默认使用的序列化器是DataContractSerializer.aspx)类。

[第八篇] 跟我一起学WCF(8)——WCF中Session、实例管理详解

WCF服务实例的管理借鉴了.NET Remoting技术的实现，同样有三种服务实例的激活方式：单调服务、会话服务和单例服务。

- 单调服务（**Percall**）：为每个客户端请求分配一个新的服务实例。类似.NET Remoting中的SingleCall模式

- 会话服务（**Persession**）：在会话期间，为每次客户端请求共享一个服务实例，类似.NET Remoting中的客户端激活模式。
- 单例服务（**Singleton**）：所有客户端请求都共享一个相同的服务实例，类似于.NET Remoting的Singleton模式。但它的激活方式需要注意一点：当为对于的服务类型进行Host的时候，与之对应的服务实例就被创建出来，之后所有的服务调用都由这个服务实例进行处理。

WCF中服务激活的默认方式是PerSession，但不是所有的Bingding都支持Session，比如BasicHttpBinding就不支持Session。你也可以通过下面的方式使ServiceContract不支持Session。

[第九篇] 跟我一起学WCF(9)——WCF回调操作的实现

在WCF中，除了支持经典的请求/应答模式外，还提供了对单向操作、双向回调操作模式的支持，此外还有流操作的支持。本文介绍在WCF中回调操作的实现。

在WCF中，并不是所有的绑定都支持回调操作，只有具有双向能力的绑定才能够用于回调。例如，HTTP本质上是与连接无关的，所以它不能用于回调，因此我们不能基于basicHttpBinding和wsHttpBinding绑定使用回调，WCF为NetTcpBinding和NetNamedPipeBinding提供了对回调的支持，因为TCP和IPC协议都支持双向通信。为了让Http支持回调，WCF提供了WsDualHttpBinding绑定，它实际上设置了两个Http通道：一个用于从客户端到服务的调用，另一个用于服务到客户端的调用。

回调操作时通过回调契约来实现的，回调契约属于服务契约的一部分，一个服务契约最多只能包含一个回调契约。一旦定义了回调契约，就需要客户端实现回调契约。在WCF中，可以通过ServiceContract的CallbackContract.aspx)属性来定义回调契约。

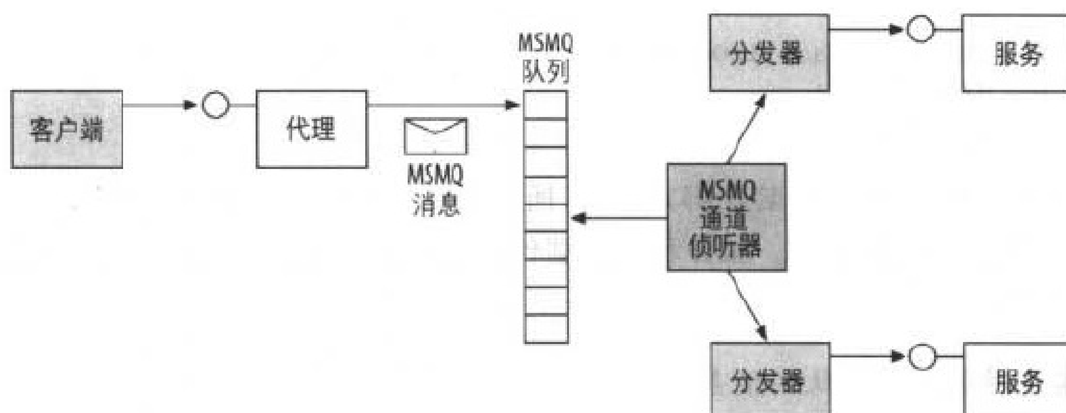
[第十篇] 跟我一起学WCF(10)——WCF中事务处理

WCF支持事务的传递，事务的传递方式由绑定的事务流属性（TransactionFlow.aspx)属性）、操作契约中的事务流选项（TransactionFlowOption.aspx)） 以及操作行为特性中的事务范围属性（TransactionScopeRequired.aspx)）共同决定。WCF事务支持的四种传播模式是：Client/Service、Client、Service和None。下图是四种传播模式对应推荐的设置。

绑定事务流	TransactionFlowOption	TransactionScopeRequired	事务模式
False	Allowed	False	None
False	Allowed	True	Service
False	NotAllowed	False	None
False	NotAllowed	True	Service
True	Allowed	False	None
True	Allowed	True	Client/Service
True	Mandatory	False	None
True	Mandatory	True	Client

[第十一篇] 跟我一起学WCF(11)——WCF中队列服务详解

既然WCF对之前多种分布式技术的继承和扩展，所以也自然支持可离线的功能，该文介绍了WCF中对队列服务的支持和实现。其实现方式与MSMQ的实现方式类似，只是WCF为队列服务提供了新的API支持，主要通过MsmqIntegrationBinding.aspx)绑定类进行支持，其通信机制如下图所示：



[第十二篇] 跟我一起学WCF(12)——WCF中Rest服务入门

由Roy Thomas Fielding 在他的博士论文（"体系结构风格和基于网络软件体系的设计"）中提出了Rest概念。Rest服务是将服务抽象为资源，每个资源都有一个唯一的统一资源标识符（URI），我们不再是通过调用操作的方式与服务进行交互了，而是通过HTTP标准动词（GET、POST、PUT和DELETE）的统一接口来完成。.NET 3.0之后，微软提供了新的API在WCF对Rest服务进行了支持，这些类包括WebHttpBinding类、WebGetAttribute、WebInvokeAttribute特性和WebServiceHost类。其实现方式和之前的WCF程序类似，只是使用新的API来对服务进行定义。

结束语：

到此，WCF系列也就告一段落了，通过对WCF技术系统的学习，我对WCF技术有了一个全面的认识，之后深入的理解就需要自己在项目中积累和实践了，希望通过这个系列也可以帮助到一些初学者。

.NET领域驱动设计实战系列

[.NET领域驱动设计实战系列]专题一：前期准备之EF CodeFirst

一、前言

从去年已经接触领域驱动设计（Domain-Driven Design）了，当时就想自己搭建一个DDD框架，所以当时看了很多DDD方面的书，例如领域驱动模式与实战，领域驱动设计：软件核心复杂性应对之道和领域驱动设计C# 2008实现等书，由于当时只是看看而已，并没有在自己代码中进行实现，只是初步了解一些DDD分层的思想和一些基本概念，例如实体，聚合根、仓储等概念，今年有机会可以去试试面试一个架构岗位的时候，深受打击，当面试官问起是否在项目中使用过DDD思想来架构项目时，我说没有，只是了解它的一些基本概念。回来之后，便重新开始学习DDD，因为我发现做成功面试一个架构师，则必须有自己的一个框架来代表自己的知识体系，而不是要你明白这个基本概念。此时学习便决定一步步来搭建一个DDD框架。但是这次的过程并不是像dax.net那样，一开始就去搭建框架，然后再用一个实际的项目来做演示框架的使用。因为我觉得这样对于一些初学者来学习的话，难度比较大，因为刚开始写框架根本看到什么，而且看dax.net的Apworks框架很多代码也不明白他为什么这么写的，从框架代码并不能看出作者怎么一步步搭建框架的，读者只能一下子看到整个成型的框架，对于刚接触DDD的朋友难度非常大，以至于学习了一段时间的DDD之后，就放弃了。这个感觉本人学习过程中深有体会。所以本系列将直接把DDD的思想应用到一个实例项目中，完全实例项目后，再从中抽取一个DDD框架出来，并且会一步步介绍如何将DDD的思想应用到一个实际项目中（dax.net中ByteartRetail项目也是直接给出一个完整的DDD演示项目的，并没有记录搭建过程，同样对于读者学习难度很大，因为一下子来吸收整个项目的知识，接受不了，读者自然就心灰意冷，也就没有继续学习DDD的动力了）。本文并没有开始介绍DDD项目的实际实现，而是一个前期准备工作，因为DDD项目中一般会使用的实体框架来完成，作为.NET阵营的人，自然首先会使EntityFramework。下面就具体介绍下EF中code First的实现，因为在后面的DDD项目实现中会使用到EF的CodeFirst。

二、EF CodeFirst的实现步骤

因为我之前没怎么接触EF的CodeFirst实现，所以在看dax.net的ByteartRetail项目的时候，对EF仓储的实现有疑惑，所以去查阅相关EF的教程发现，原来应用了EF中的CodeFirst。所以把过程记录下来。下面就具体介绍下使用EF CodeFirst的具体实现步骤。

步骤一：创建一个**Asp.net MVC 4 Web**项目，创建成功后，再添加一个**Model**类

CodeFirst自然是先写实体类了，这里演示的是一个Book实体类，具体类的实现代码如下：

```
public class Book
    {
        public int BookId { get; set; }
        public string BookName { get; set; }
        public string Author { get; set; }
        public string Publisher { get; set; }
        public decimal Price { get; set; }
        public string Remark { get; set; }
    }
```

将使用这个类表示数据库中的一个表，每个Book类的实例对应数据库中的一行，Book类中的每个属性映射为数据库中的一列。

步骤二：创建**“BookDbContext”**的类

使用Nuget安装Entity Framework，安装成功后，在Models文件夹下新建一个“BookDbContext”的类，将类派生自“DbContext”类（命名空间为System.Data.Entity，dll在EntityFramework），具体BookDbContext的实现如下：

BookDbContext代表EF中Book在数据库中的上下文对象，通过DbSet<Book>使实体类与数据库关联起来。Books属性表示数据中的数据集实体，用来处理数据的存取与更新。

步骤三：添加数据库连接

在Web.config文件中，修改数据库连接字符串的配置，这里将数据库连接的name属性设置为BookDbContext，后面代码将会使用到该名字，并根据连接创建相应的数据库。

步骤四：为**Book**创建控制器和**Index**视图

首先创建一个控制器：在"Controlllers"上右键>添加>控制器，在打开的添加控制器对话框中，将控制器的名称改为"BookController"，模板选择”空控制器“。修改BookController的代码为如下所示：

```
public class BookController : Controller
    {
        readonly BookDbContext _db = new BookDbContext();

        //
        // GET: /Book/

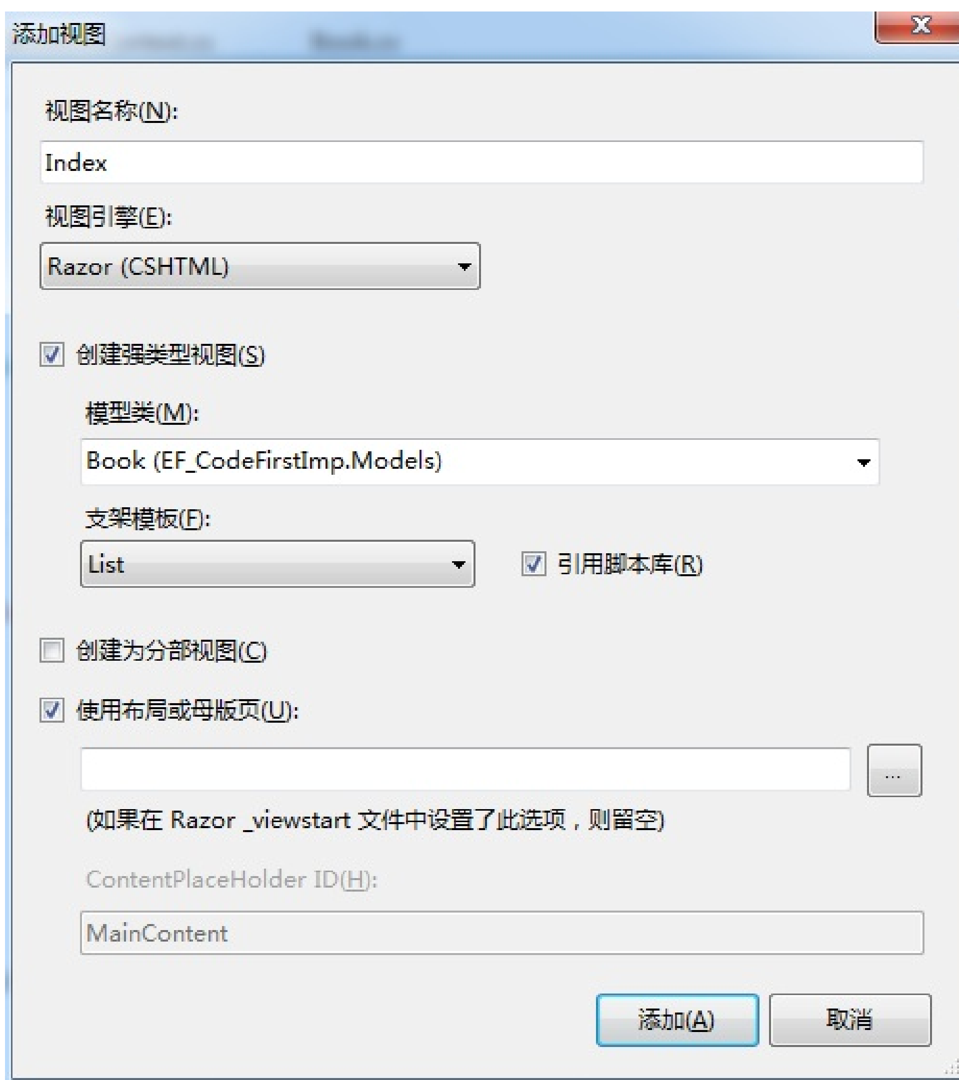
        public ActionResult Index()
        {

            var books = from b in _db.Books
                        where b.Author == "Learninghard"
                        select b;

            return View(books.ToList());
        }

        public ActionResult Create()
        {
            return View();
        }
    }
```

在Index方法内右键>"添加视图"，在打开的"添加视图"对话框，勾选"创建强类型视图"，在模型类列表中选择"Book"（如果选择列表为空，则需要首先编译下项目），在支架模板列表中选择"List"，具体如下图所示：



点击添加按钮，VS为我们创建了Index.cshtml文件，修改Index.cshtml代码为如下所示的代码：

```
@model IEnumerable<EF_CodeFirstImp.Models.Book>

@{
    ViewBag.Title = "图书列表-EF CodeFirstImp";
}

<h2>Index</h2>

<p>
    @Html.ActionLink("添加图书", "Create")
</p>
<table>
    <tr>
        <th>
           图书名称
        </th>
        <th>
            作者
        </th>
```

```
        <th>
           出版社
        </th>
        <th>
            价格
        </th>
        <th>
            备注
        </th>
        <th></th>
    </tr>

@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.BookName)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Author)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Publisher)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Price)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Remark)
        </td>
        <td>
            @Html.ActionLink("编辑", "Edit", new { id=item.BookId }
            @Html.ActionLink("查看", "Details", new { id=item.BookI
            @Html.ActionLink("删除", "Delete", new { id=item.BookId
        </td>
    </tr>
}

</table>
```

编译并运行程序，在浏览器中输入地址：http://localhost:2574/Book，得到的运行结果如下：



尽管没有数据，但EF已经为我们创建了相应的数据库了。此时在App_Data文件夹下生成了BookDB数据库，在解决方案点击选择所有文件，将BookDB数据库包括在项目中。

步骤五：添加**Create**视图

在BookController中的Create方法右键添加视图来添加Create视图，此时模型类仍然选择Book，但支架模板选择"Create"。添加成功后，VS会在Views/Book目录下添加一个Create.cshtml文件，由于这里选择了Create支架框架，所以VS会为我们生成一些默认的代码。在这个视图模板中，指定了强类型Book作为它的模型类，VS检查Book类，并根据Book类的属性，生成对应的标签名和编辑框，我们修改标签使其显示中文，修改会的代码如下所示：

```
@model EF_CodeFirstImp.Models.Book

@{
    ViewBag.Title = "添加图书";
}

<h2>添加图书</h2>

@using (Html.BeginForm()) {
    @Html.AntiForgeryToken()
    @Html.ValidationSummary(true)

    <fieldset>
        <legend>图书</legend>

        <div class="editor-label">
            图书名称：
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.BookName)
```

```
            @Html.ValidationMessageFor(model => model.BookName)
        </div>

        <div class="editor-label">
            作者：
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Author)
            @Html.ValidationMessageFor(model => model.Author)
        </div>

        <div class="editor-label">
           出版社：
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Publisher)
            @Html.ValidationMessageFor(model => model.Publisher)
        </div>

        <div class="editor-label">
            价格：
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Price)
            @Html.ValidationMessageFor(model => model.Price)
        </div>

        <div class="editor-label">
           备注：
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Remark)
            @Html.ValidationMessageFor(model => model.Remark)
        </div>

        <p>
            <input type="submit" value="添加" />
        </p>
    </fieldset>
}

<div>
    @Html.ActionLink("Back to List", "Index")
</div>

@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
}
```

分析上面的代码：

- @model EF_CodeFirstImp.Models.Book：指定该视图模板中的“模型”强类型

化是一个Book类。

- @using (Html.BeginForm()){ }：创建一个Form表单，在表单中包含了对于Book类所生成的对应字段。
- @Html.EditorFor(model => model.BookName)：根据模型生成模型中BookName的编辑控件（生成一个Input元素）
- @Html.ValidationMessageFor(model => model.BookName)：根据模型生成模型中BookName的验证信息。



步骤六：添加**Create**的**Postback**方法

```
public class BookController : Controller
    {
        readonly BookDbContext _db = new BookDbContext();

        //
        // GET: /Book/

        public ActionResult Index()
        {
            var books = from b in _db.Books
                        where b.Author == "Learninghard"
                        select b;

            return View(books.ToList());
        }

        public ActionResult Create()
        {
            return View();
        }

        [HttpPost]
        public ActionResult Create(Book book)
        {
            if (ModelState.IsValid)
            {
                _db.Books.Add(book);
                _db.SaveChanges();
                return RedirectToAction("Index");
            }
            else
                return View(book);
        }

    }
```

这时，我们在添加图书界面中输入数据，并点击"添加"按钮时，数据库中就会添加一行记录。打开数据库，我们将看到如下截图的数据：

	BookId	BookName	Author	Publisher	Price	Remark
▶	1	Learninghard ...	李志	人民邮电出版社	41.60	全面的C#学习...
*	NULL	NULL	NULL	NULL	NULL	NULL

步骤七：设置视图模型的数据验证

我们可以在模型类中显式地追加一个验证规则，使得对输入数据进行验证。修改之前的Book类为如下：

```
using System.ComponentModel.DataAnnotations; //需要额外添加该命名空间
public class Book
    {
        public int BookId { get; set; }
        [Required(ErrorMessage = "必须输入图书名称")]
        [StringLength(maximumLength: 100, MinimumLength = 1, ErrorM
        public string BookName { get; set; }
        [Required(ErrorMessage ="必须输入作者名称")]
        public string Author { get; set; }
        [Required(ErrorMessage ="必须输入出版社名称" )]
        public string Publisher { get; set; }
        public decimal Price { get; set; }
        public string Remark { get; set; }
    }
```

此时重新运行，并打开添加图书页面，当不输入任何数据的时候，点击"添加"按钮时，界面会出现一些提示信息，并阻止我们进行数据的提交操作，具体的结果界面如下所示：

将你的徽标放置在此处

注册 登录

主页 关于 联系方式

添加图书

图书名称：

必须输入图书名称

作者：

必须输入作者名称

出版社：

必须输入出版社名称

价格：

Price 字段是必需的。

备注：

添加

Back to List

另外，EF创建数据库除了在第三步中添加连接字符串的方式外，还可以定义defaultConnectionFactory中添加parameters节点的方式来完成，dax.net中ByteartRetail项目中就是采用了这种方式。下面注释掉connectionStrings节点，在defaultConnectionFactory添加如下parameters节点：

```
<entityFramework>
    **<defaultConnectionFactory type="System.Data.Entity.Infrastruc
    <parameters>
      <parameter value="Data Source=(LocalDb)\v11.0;Initial Catalog
    </parameters>
    </defaultConnectionFactory>**
     <providers>
      <provider invariantName="System.Data.SqlClient" type="System.
    </providers>
  </entityFramework>
```

entityFramework节点是使用Nuget添加Entity Framework后自动添加的节点。下面测试下这种方案是否可以成功生成**BookDB_2**数据库呢？

运行项目，在浏览器中输入地址：http://localhost:2574/Book，显示界面成功后，你将在你的App_Data目录下看到如下截图：

BookDB.mdf	2015/4/26 12:47	SQL Server Data...	3,136 KB
BookDB_2.mdf	2015/4/26 12:53	SQL Server Data...	2,112 KB
BookDB_2_log.ldf	2015/4/26 12:53	SQL Server Data...	1,024 KB
BookDB_log.ldf	2015/4/26 12:47	SQL Server Data...	1,024 KB

从上图可以发现，这种方式同样成功生成了数据库。

三、总结

到这里，领域驱动设计实战系列的前期准备就结束了，本文主要介绍了如何使用EF CodeFirst的功能自动生成数据库、以及实体的添加、查看操作等。这里也简单介绍了MVC相关内容。下一专题将介绍如何利用DDD的思想来构建一个简单的网站，接下来的系列就逐一加入DDD的概念来对该网站进行完善。

本文所有源码下载：EFCodeFirstImp.zip

[.NET领域驱动设计实战系列]专题二：结合领域驱动设计的面向服务架构来搭建网上书店

一、前言

在前面专题一中，我已经介绍了我写这系列文章的初衷了。由于dax.net中的DDD框架和Byteart Retail案例并没有对其形成过程做一步步分析，而是把整个DDD的实现案例展现给我们，这对于一些刚刚接触领域驱动设计的朋友可能会非常迷茫，从而觉得领域驱动设计很难，很复杂，因为学习中要消化一个整个案例的知识，这样未免很多人消化不了就打退堂鼓，就不继续研究下去了，所以这样也不利于DDD的推广。然而本系列可以说是刚接触领域驱动设计朋友的福音，本系列将结合领域驱动设计的思想来一步步构建一个网上书店，从而让大家学习DDD不再枯燥和可以看到一个DDD案例的形成历程。最后，再DDD案例完成之后，将从中抽取一个领域驱动的框架，从而大家也可以看到一个DDD框架的形成历程，这样就不至于一下子消化一整个框架和案例的知识，而是一步步消化。接下来，该专题将介绍的是：结合领域驱动设计的SOA架构来构建网上书店，本专题中并没有完成网上书店的所有页面和覆盖DDD中的所有内容，而只是一部分，后面的专题将会在本专题的网上书店进行一步步完善，通过一步步引入DDD的内容和重构来完成整个项目。

二、DDD分层架构

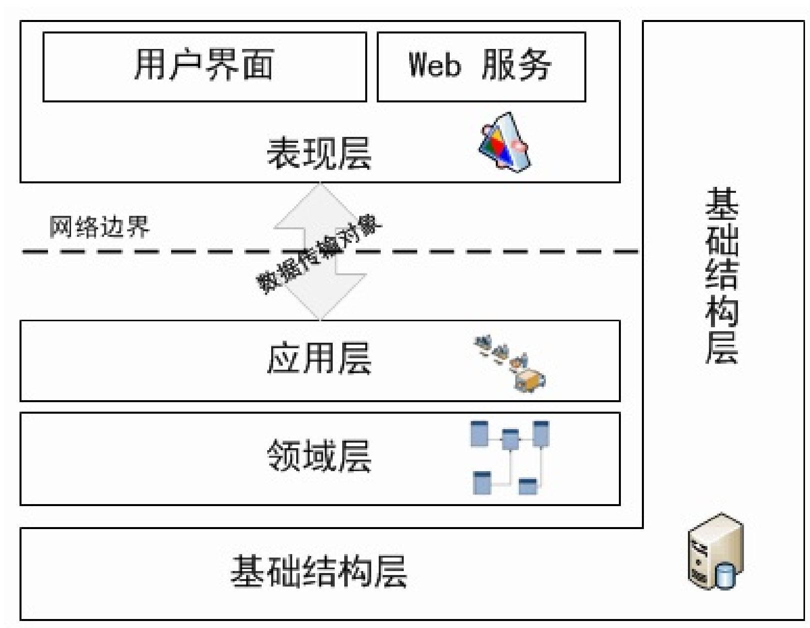
从概念上说，领域驱动设计架构主要分为四层，分别为：基础设施层、领域层、应用层和表现层。

- 基础结构层：该层专为其他各层提供各项通用技术框架支持。像一些配置文件处理、缓存处理，事务处理等都可以放在这里。
- 领域层：简单地说就是业务所涉及的领域对象（包括实体、值对象）、领域服务等。该层就是所谓的领域模型了，领域驱动设计提倡是富领域模型，富领域模型指的是：尽量将业务逻辑放在归属于它的领域对象中。而之前的三层架构中的领域模型都是贫血领域模型，因为在三层中的领域模型只包含业务属性，而不包含任何业务逻辑。本专题的网上书店领域模型目前还没有包含任何业务逻辑，在后期将会完善。

实体可以认为对应于数据库的表，而值对象一般定义在实体类中。

- 应用层：该层不包含任何领域逻辑，它主要用来对任务进行协调，它构建了表现层和领域层的桥梁。SOA架构就是在该层进行实现的。
- 表现层：指的是用户界面，例如Asp.net mvc网站，WPF、Winform和控制台等。它主要用来想用户展现内容。

下面用一个图来形象展示DDD的分层架构：



本系列介绍的领域驱动设计实战，则自然少了领域驱动设计分层架构的实现了，上面简单介绍了领域驱动的分层架构，接下来将详细介绍在网上书店中各层是如何去实现的。

三、网上书店领域模型层的实现

在应用领域驱动设计的思想来构建一个项目，则第一步就是了解需求，明白项目的业务逻辑，了解清楚业务逻辑后，则把业务逻辑抽象成领域对象，领域对象所放在的位置也就是领域模型层了。该专题介绍的网上书店主要完成了商品所涉及的页面，包括商品首页，单个商品的详细信息等。所以这里涉及的领域实体包括2个，一个是商品类，另外一个就是类别类，因为在商品首页中，需要显示所有商品的类别。在给出领域对象的实现之前，这里需要介绍领域层中所涉及的几个概念。

- 聚合根：聚合根也是实体，但与实体不同的是，聚合根是由实体和值对象组成的系统边界对象。举个例子来说，例如订单和订单项，根据业务逻辑，我们需要跟踪订单和订单项的状态，所以设计它们都为实体，但只有订单才是聚合根对象，而订单项不是，因为订单项只有在订单中才有意义，意思就是说：用户不能直接看到订单项，而是先查询到订单，然后再看到该订单下的订单项。所以聚合根可以理解为用户直接操作的对象。在这里商品类和类别类都是一个聚合根。

根据面向接口编程原则，我们在领域模型中应该定义一个实体接口和聚合根接口，而因为聚合根也是属于实体，所以聚合根接口继承于实体接口，而商品类和类别类都是聚合根，所以它们都实现聚合根接口。如果像订单项只是实体不是聚合根的类则实现实体接口。有了上面的分析，则领域模型层的实现也就自然出来了，下面是领域对象的具体实现：

```
// 商品类
    public class Product : AggregateRoot
    {
        public string Name { get; set; }

        public string Description { get; set; }

        public decimal UnitPrice { get; set; }

        public string ImageUrl { get; set; }

        public bool IsNew{ get; set; }

        public override string ToString()
        {
            return Name;
        }
    }
```

```
// 类别类
    public class Category : AggregateRoot
    {
        public string Name { get; set; }

        public string Description { get; set; }

        public override string ToString()
        {
            return this.Name;
        }
    }
```

另外，领域层除了实现领域对象外，还需要定义仓储接口，而仓储层则是对仓储接口的实现。仓储可以理解为在内存中维护一系列聚合根的集合，而聚合根不可能一直存在于内存中，当它不活动时会被持久化到数据中。而仓储层完成的任务是持久化聚合根对象到数据或从数据库中查询存储的对象来重新创建领域对象。

仓储层有几个需要明确的概念：

1. 仓储里面存放的对象一定是聚合根，因为领域模型是以聚合根的概念去划分的，聚合根就是我们操作对象的一个边界。所以我们都是对某个聚合根进行操作的，而不存在对聚合内的值对象进行操作。因此，仓储只针对聚合根设计。
2. 因为仓储只针对聚合根设计，所以一个聚合根需要实现一个仓储。
3. 不要把仓储简单理解为DAO，仓储属于领域模型的一部分，代表了领域模型向外界提供接口的一部分，而DAO是表示数据库向上层提供的接口表示。一个是针对领域模型而言，而另一个针对数据库而言。两者侧重点不一样。
4. 仓储分为定义部分和实现部分，在领域模型中定义仓储的接口，而在基础设施层实现具体的仓储。这样做的原因是：由于仓储背后的实现都是在和数据库打

交道，但是我们又不希望客户（如应用层）把重点放在如何从数据库获取数据的问题上，因为这样做会导致客户（应用层）代码很混乱，很可能会因此而忽略了领域模型的存在。所以我们需要提供一个简单明了的接口，供客户使用，确保客户能以最简单的方式获取领域对象，从而可以让它专心的不会被什么数据访问代码打扰的情况下协调领域对象完成业务逻辑。这种通过接口来隔离封装变化的做法其实很常见。由于客户面对的是抽象的接口并不是具体的实现，所以我们可以随时替换仓储的真实实现，这很有助于我们做单元测试。在本专题的案例中，我们把仓储层的实现单独从基础设施层拎出来了，作为一个独立的层存在。这也就是为什么DDD分层中没有定义仓储层啊，而本专题的案例中多了一个仓储层的实现。
5. 仓储在设计查询接口时，会经常用到规约模式（Specification Pattern）。本专题的案例中没有给出，这点将会在后面专题添加上去。
6. 仓储一般不负责事务处理，一般事务处理会交给“工作单元（Unit Of Work）”去处理，同样本专题也没有涉及工作单元的实现，这点同样会在后面专题继续完善。这里列出来让大家对后面的专题可以有个清晰的概念，而不至于是空穴来风的。

介绍完仓储之后，接下来就在领域层中定义仓储接口，因为本专题中涉及到2个聚合根，则自然需要实现2个仓储接口。根据面向接口编程原则，我们让这2个仓储接口都实现与一个公共的接口：IRepository接口。另外仓储接口还需要定义一个仓储上下接口，因为在Entity Framework中有一个DbContex类，所以我们需要定义一个EntityFramework上下文对象来对DbContex进行包装。也就自然有了仓储上下文接口了。经过上面的分析，仓储接口的实现也就一目了然了。

```
// 仓储接口
    public interface IRepository<TAggregateRoot>
        where TAggregateRoot :class, IAggregateRoot
    {
        void Add(TAggregateRoot aggregateRoot);

        IEnumerable<TAggregateRoot> GetAll();

        // 根据聚合根的ID值，从仓储中读取聚合根
        TAggregateRoot GetByKey(Guid key);
    }
```

这样我们就完成了领域层的搭建了，接下面，我们就需要对领域层中定义的仓储接口进行实现了。我这里将仓储接口的实现单独弄出了一个层，当然你也可以放在基础设施层中的Repositories文件夹中。不过我看很多人都直接拎出来的。我这里也是直接作为一个层。

四、网上书店Repository（仓储）层的实现

定义完仓储接口之后，接下来就是在仓储层实现这些接口，完成领域对象的序列化。首先是产品仓储的实现：

```
// 商品仓储的实现
```

```
    public class ProductRepository : IProductRepository
    {
        #region Private Fields

        private readonly IEntityFrameworkRepositoryContext _efConte

        #endregion
        #region Public Properties

        public IEntityFrameworkRepositoryContext EfContext
        {
            get { return this._efContext; }
        }

        #endregion

        #region Ctor

        public ProductRepository(IRepositoryContext context)
        {
            var efContext = context as IEntityFrameworkRepositoryCo
            if (efContext != null)
                this._efContext = efContext;
        }

        #endregion
        public IEnumerable<Product> GetNewProducts(int count = 0)
        {
            var ctx = this.EfContext.DbContex as OnlineStoreDbConte
            if (ctx == null)
                return null;
            var query = from p in ctx.Products
                        where p.IsNew == true
                        select p;
            if (count == 0)
                return query.ToList();
            else
                return query.Take(count).ToList();
        }

        public void Add(Product aggregateRoot)
        {
            throw new NotImplementedException();
        }

        public IEnumerable<Product> GetAll()
        {
            var ctx = this.EfContext.DbContex as OnlineStoreDbConte
            if (ctx == null)
                return null;
            var query = from p in ctx.Products
                        select p;
            return query.ToList();
```

```
        }

        public Product GetByKey(Guid key)
        {
            return EfContext.DbContex.Products.First(p => p.Id == k
        }
    }
```

接下来是类别仓储的实现：

```
// 类别仓储的实现
    public class CategoryRepository :ICategoryRepository
    {
        #region Private Fields

        private readonly IEntityFrameworkRepositoryContext _efConte

        public CategoryRepository(IRepositoryContext context)
        {
            var efContext = context as IEntityFrameworkRepositoryC
            if (efContext != null)
                this._efContext = efContext;
        }

        #endregion
        #region Public Properties

        public IEntityFrameworkRepositoryContext EfContext
        {
            get { return this._efContext; }
        }

        #endregion

        public void Add(Category aggregateRoot)
        {
            throw new System.NotImplementedException();
        }

        public IEnumerable<Category> GetAll()
        {
            var ctx = this.EfContext.DbContex as OnlineStoreDbConte
            if (ctx == null)
                return null;
            var query = from c in ctx.Categories
                        select c;
            return query.ToList();
        }

        public Category GetByKey(Guid key)
        {
            return this.EfContext.DbContex.Categories.First(c => c.
        }
    }
```

由于后期除了实现基于EF仓储的实现外，还想实现基于MongoDb仓储的实现，所以在仓储层中创建了一个EntityFramework的文件夹，并定义了一个IEntityFrameworkRepositoryContext接口来继承于IRepositoryContext接口，由于EF中持久化数据主要是由DbContext对象来完成了，为了有自己框架模型，我在这

里定义了OnlineStoreDbContext来继承DbContext，从而用OnlineStoreDbContext来对DbContext进行了一次包装。经过上面的分析之后，接下来对于实现也就一目了然了。首先是OnlineStoreDbContext类的实现：

```
public sealed class OnlineStoreDbContext : DbContext
    {
        #region Ctor
        public OnlineStoreDbContext()
            : base("OnlineStore")
        {
            this.Configuration.AutoDetectChangesEnabled = true;
            this.Configuration.LazyLoadingEnabled = true;
        }
        #endregion

        #region Public Properties

        public DbSet<Product> Products
        {
            get { return this.Set<Product>(); }
        }

        public DbSet<Category> Categories
        {
            get { return this.Set<Category>(); }
        }

        // 后面会继续添加属性，针对每个聚合根都会定义一个DbSet的属性
        // ...
        #endregion
    }
```

接下来就是IEntityFrameworkRepositoryContext接口的定义以及它的实现了。具体代码如下所示：

```
 public class EntityFrameworkRepositoryContext : IEntityFrameworkRe
    {
        // 引用我们定义的OnlineStoreDbContext类对象
        public OnlineStoreDbContext DbContex
        {
            get { return new OnlineStoreDbContext(); }
        }
    }
```

这样，我们的仓储层也就完成了。接下来就是应用层的实现。

五、网上书店应用层的实现

应用层应用了面向服务结构进行实现，采用了微软面向服务的实现WCF来完成的。网上书店的整个架构完全遵循着领域驱动设计的分层架构，用户通过UI层（这里实现的是Web页面）来进行操作，然后UI层调用应用层来把服务进行分发，通过调用基础设施层中仓储实现来对领域对象进行持久化和重建。这里应用层主要采用WCF来实现的，其中引用了仓储接口。针对服务而言，首先就需要定义服务契约了，这里我把服务契约的定义单独放在了一个服务契约层，当然你也可以在应用层中创建一个服务契约文件夹。首先就去看看服务契约的定义：

```
// 商品服务契约的定义
    [ServiceContract(Namespace="")]
    public interface IProductService
    {
        #region Methods
        // 获得所有商品的契约方法
        [OperationContract]
        IEnumerable<Product> GetProducts();

        // 获得新上市的商品的契约方法
        [OperationContract]
        IEnumerable<Product> GetNewProducts(int count);

        // 获得所有类别的契约方法
        [OperationContract]
        IEnumerable<Category> GetCategories();

        // 根据商品Id来获得商品的契约方法
        [OperationContract]
        Product GetProductById(Guid id);

        #endregion
    }
```

接下来就是服务契约的实现，服务契约的实现我放在应用层中，具体的实现代码如下所示：

```
// 商品服务的实现
    public class ProductServiceImp : IProductService
    {
        #region Private Fields
        private readonly IProductRepository _productRepository;
        private readonly ICategoryRepository _categoryRepository;
        #endregion

        #region Ctor
        public ProductServiceImp(IProductRepository productReposit
        {
            _categoryRepository = categoryRepository;
            _productRepository = productRepository;
        }

        #endregion

        #region IProductService Members
        public IEnumerable<Product> GetProducts()
        {
            return _productRepository.GetAll();
        }

        public IEnumerable<Product> GetNewProducts(int count)
        {
            return _productRepository.GetNewProducts(count);
        }

        public IEnumerable<Category> GetCategories()
        {
            return _categoryRepository.GetAll();
        }

        public Product GetProductById(Guid id)
        {
            var product = _productRepository.GetByKey(id);
            return product;
        }

        #endregion
    }
```

最后就是创建WCF服务来调用服务契约实现了。创建一个后缀为.svc的WCF服务文件，WCF服务的具体实现如下所示：

```
// 商品WCF服务
    public class ProductService : IProductService
    {
        // 引用商品服务接口
        private readonly IProductService _productService;

        public ProductService()
        {
            _productService = ServiceLocator.Instance.GetService<IP
        }

        public IEnumerable<Product> GetProducts()
        {
            return _productService.GetProducts();
        }

        public IEnumerable<Product> GetNewProducts(int count)
        {
            return _productService.GetNewProducts(count);
        }

        public IEnumerable<Category> GetCategories()
        {
            return _productService.GetCategories();
        }

        public Product GetProductById(Guid id)
        {
            return _productService.GetProductById(id);
        }
    }
```

到这里我们就完成了应用层面向服务架构的实现了。从商品的WCF服务实现可以看到，我们有一个ServiceLocator的类。这个类的实现采用服务定位器模式，关于该模式的介绍可以参考dax.net的服务定位器模式的介绍。该类的作用就是调用方具体的实例，简单地说就是通过服务接口定义具体服务接口的实现，将该实现返回给调用者的。这个类我这里放在了基础设施层来实现。目前基础设施层只有这一个类的实现，后期会继续添加其他功能，例如缓存功能的支持。

另外，在这里使用了Unity依赖注入容器来对接口进行注入。主要的配置文件如下所示：

```
<configuration>
  <configSections>
    <section name="entityFramework" type="System.Data.Entity.Intern
    <section name="unity" type="Microsoft.Practices.Unity.Configura
  </configSections>

  <!-- Entity Framework 配置信息-->
```

```
  <entityFramework>
    <defaultConnectionFactory type="System.Data.Entity.Infrastructu
      <parameters>
        <parameter value="Data Source=(LocalDb)\v11.0; Initial Cata
      </parameters>
    </defaultConnectionFactory>
  </entityFramework>

  <!--Unity的配置信息-->
  <unity xmlns="http://schemas.microsoft.com/practices/2010/unity">
    <container>
      <!--仓储接口的注册-->

      <register type="OnlineStore.Domain.Repositories.IRepositoryCo
      <register type="OnlineStore.Domain.Repositories.IProductRepos
      <register type="OnlineStore.Domain.Repositories.ICategoryRepo

      <!--应用服务的注册-->
      <register type="OnlineStore.ServiceContracts.IProductService,
    </container>
  </unity>

  <appSettings>
    <add key="aspnet:UseTaskFriendlySynchronizationContext" value="
  </appSettings>
  <system.web>
    <compilation debug="true" targetFramework="4.5"/>
    <httpRuntime targetFramework="4.5.1"/>
  </system.web>
  <!--WCF 服务的配置信息-->
  <system.serviceModel>
    <behaviors>
      <serviceBehaviors>
        <behavior name="">
          <serviceMetadata httpGetEnabled="true" httpsGetEnabled="t
          <serviceDebug includeExceptionDetailInFaults="true"/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
    <services>
      <service name="OnlineStore.Application.ServiceImplementations
        <endpoint address="" binding="wsHttpBinding" contract="Onli
        <!--<endpoint contract="IMetadataExchange" binding="mexHttp
      </service>
    </services>

    <serviceHostingEnvironment aspNetCompatibilityEnabled="true" mu
  </system.serviceModel>
  <system.webServer>
    <modules runAllManagedModulesForAllRequests="true"/>
    <!--
        To browse web app root directory during debugging, set the
        Set to false before deployment to avoid disclosing web app
```

```
        -->
      <directoryBrowse enabled="true"/>
    </system.webServer>
  </configuration>
```

六、基础设施层的实现

基础设施层在本专题中只包含了服务定位器的实现，后期会继续添加对其他功能的支持，ServiceLocator类的具体实现如下所示：

```
   // 服务定位器的实现
      public class ServiceLocator : IServiceProvider
      {
          private readonly IUnityContainer _container;
          private static ServiceLocator _instance = new ServiceLocato

          private ServiceLocator()
          {
              _container = new UnityContainer();
              _container.LoadConfiguration();
          }

          public static ServiceLocator Instance
          {
              get { return _instance; }
          }

          #region Public Methods

          public T GetService<T>()
          {
              return _container.Resolve<T>();
          }

          public IEnumerable<T> ResolveAll<T>()
          {
              return _container.ResolveAll<T>();
          }

          public T GetService<T>(object overridedArguments)
          {
              var overrides = GetParameterOverrides(overridedArgument
              return _container.Resolve<T>(overrides.ToArray());
          }

          public object GetService(Type serviceType, object overridec
          {
              var overrides = GetParameterOverrides(overridedArgument
              return _container.Resolve(serviceType, overrides.ToArra
```

```
        }

        #endregion

        #region Private Methods
        private IEnumerable<ParameterOverride> GetParameterOverride
        {
            var overrides = new List<ParameterOverride>();
            var argumentsType = overridedArguments.GetType();
            argumentsType.GetProperties(BindingFlags.Public | Bindi
                .ToList()
                .ForEach(property =>
                {
                    var propertyValue = property.GetValue(overridec
                    var propertyName = property.Name;
                    overrides.Add(new ParameterOverride(propertyNar
                });
            return overrides;
        }
        #endregion

        #region IServiceProvider Members

        public object GetService(Type serviceType)
        {
            return _container.Resolve(serviceType);
        }

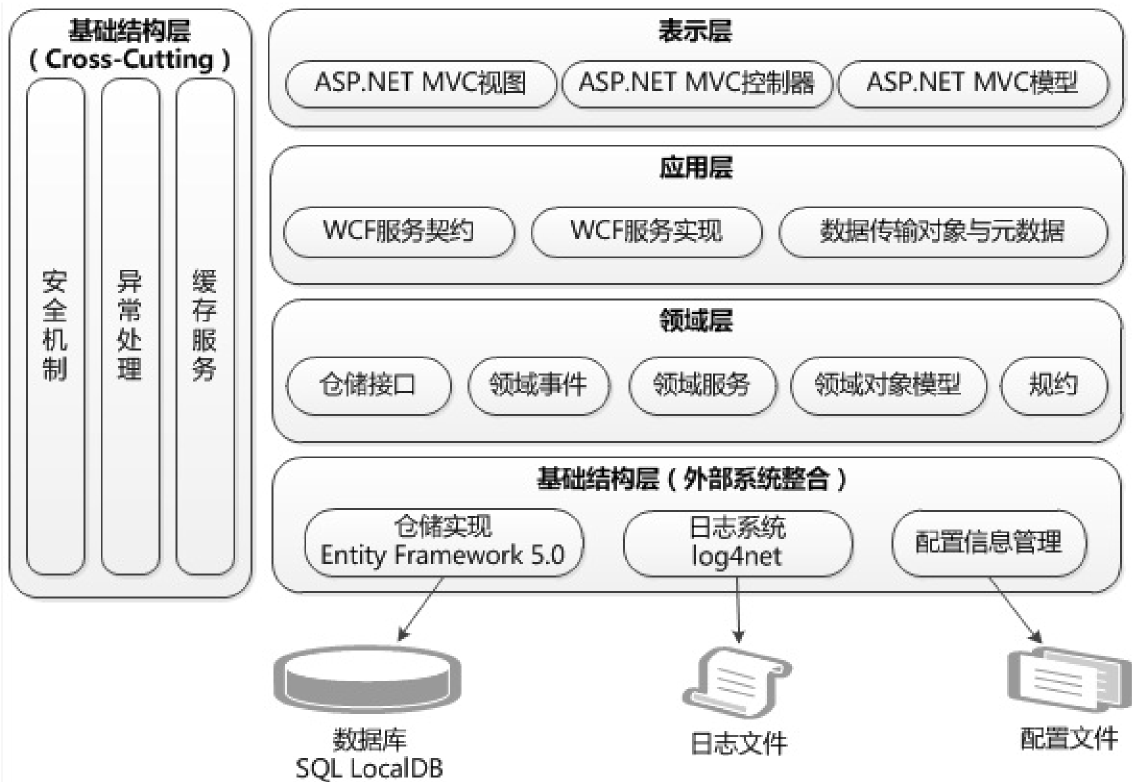
        #endregion
    }
```

七、UI层的实现

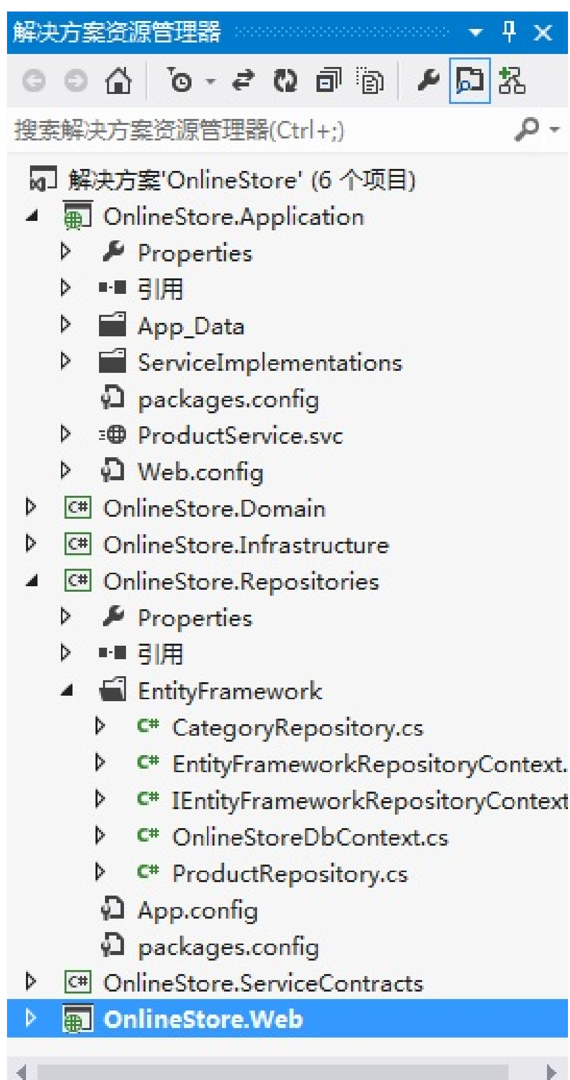
根据领域驱动的分层架构，接下来自然就是UI层的实现了，这里UI层的实现采用Asp.net MVC 技术来实现的。UI层主要包括商品首页的实现，和详细商品的实现，另外还有一些附加页面的实现，例如，关于页面，联系我们页面等。关于UI层的实现，这里就不一一贴出代码实现了，大家可以在最后的源码链接自行下载查看。

八、系统总体架构

经过上面的所有步骤，本专题中的网上书店构建工作就基本完成了，接下来我们来看看网上书店的总体架构图（这里架构图直接借鉴了dax.net的图了，因为本系列文章也是对其Byteart Retail项目的剖析过程）：



最后附上整个解决方案的结构图：



九、网上书店运行效果

实现完之后，大家是不是都已经迫不及待地想看到网上书店的运行效果呢？下面就为大家来揭晓，目前网上书店主要包括2个页面，一个是商品首页的展示和商品详细信息的展示。首先看下商品首页的样子吧：

图书的详细信息页面：

十、总结

到这里，本专题的内容就介绍完了， 本专题主要介绍面向领域驱动设计的分层架构和面向服务架构。然后结合它们在网上书店中进行实战演练。在后面的专题中我会在该项目中一直进行完善，从而形成一个完整了DDD案例。在接下来的专题会对仓储的实现应用规约模式，在应用之前，我会先写一个专题来介绍规约模式来作为一个准备工作。

GitHub 开源地址：https://github.com/lizhi5753186/OnlineStore。

[.NET领域驱动设计实战系列]专题三：前期准备之规约模式(Specification Pattern)

一、前言

在专题二中已经应用DDD和SOA的思想简单构建了一个网上书店的网站，接下来的专题中将会对该网站补充更多的DDD的内容。本专题作为一个准备专题，因为在后面一个专题中将会网上书店中的仓储实现引入规约模式。本专题将详细介绍了规约模式。

二、什么是规约模式

讲到规约模式，自然想到的是什么是规约模式呢？从名字上看，规约模式就是一个约束条件，我们在使用仓储进行查询的时候，这时候就会牵涉到很多查询条件，例如名字包含C#的书名等条件。这样就自然需要引入规约模式了。规约模式的作用可以自由组装业务逻辑元素。Specification类有一个IsSatisifiedBy函数，用于校验某个对象是否满足该Specification所表达的条件。多个Specification对象可以组装起来，生成新的Specification对象，这样可以通过组装的方式来定制新的条件。简单地说，规约模式就是对查询条件表达式用类的形式进行封装。那这样的话，规约模式引入有什么作用呢？

三、为什么需要引入规约模式模式

上面只是简单介绍了规约模式的作用——可以自由组装业务逻辑元素。这样文字表述未免枯燥了点，下面通过一个具体例子来说明下。

对于在仓储中，我们经常会定义下面的接口

接下来就是实现这个接口，并在类中分别实现接口中的方法。这样设计的好处就是一目了然，可以方便地看到Product仓储到底提供了哪些功能。

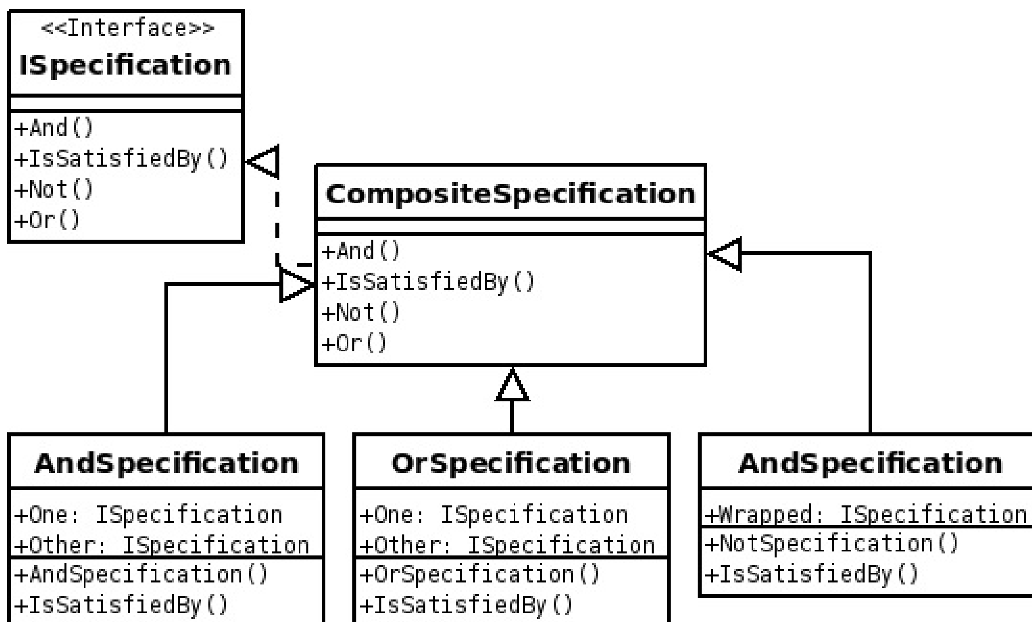
对于这种设计，对于简单系统并且今后扩展的可能性不大，那么这样的设计非常合适，因为其简洁高效。但如果你正在设计一个中大型系统，那么，针对上面的设计，你就需要考虑下面的问题了：

1. 今后如果需要添加新的查询逻辑，结果一大堆相关代码都需要修改，上面的设计能便于扩展吗？
2. 由于业务的扩展，上面的设计会导致接口变得越来越大，团队成员可能会对这个接口进行修改，添加新的接口方法。

规约模式就是DDD引入解决上面问题的一种模式。下面让我们来看看规约模式的定义与实现。

四、规约模式的传统实现

首先来看下规约模式的类结构图：



上图是摘自维基百科里面的，通过设计图我们很容易实现规约模式，这样之所以称为的传统实现，因为后面会对该实现应用C#的特性来对该实现进行简化，使其更加简单轻量。首先我们需要定义一个ISpecification接口，在接口中定义四个方法：And、Not、Or和IsSatifiedBy方法，具体接口的定义如下所示：

```
// 规约接口的定义
    public interface ISpecification<T>
    {
        bool IsSatisfiedBy(T candidate);
        ISpecification<T> And(ISpecification<T> specification);
        ISpecification<T> Or(ISpecification<T> specification);
        ISpecification<T> Not(ISpecification<T> specification);
    }
```

实现了ISpeification的对象意味着是一个Specification，即一种筛选条件，我们可以与其他Specification对象通过And、Or和Not操作来生成新的逻辑，即组合成新的筛选条件，为了方便“组合逻辑”的实现，这里还需要定义一个抽象的CompositeSpecification类：

```
// 因为And,OR和Not方法在所有的Specification都需要实现，只有IsSatisfiedBy
    // 所以为了复用，定义一个抽象类来实现And,Or和And操作,并且留IsSatisfied
    public abstract class CompositeSpecification<T>: ISpecificatior
    {
        public abstract bool IsSatisfiedBy(T candidate);

        public ISpecification<T> And(ISpecification<T> specificatic
        {
            return new AndSpecification<T>(this, specification);
        }

        public ISpecification<T> Or(ISpecification<T> specificatior
        {
            return new OrSpecification<T>(this, specification);
        }

        public ISpecification<T> Not(ISpecification<T> specificatic
        {
            return new NotSpecification<T>(specification);
        }
    }
```

CompositeSpecification提供了构建符合Specification的基础逻辑，它提供了And、Or和Not方法的实现，让其他Specification类只需要专注于IsSatisfiedBy方法的实现即可（这里有点模板方法模式的影子）。下面是And、Or和Not规约的具体实现：

```
// AndSpecification,OrSpecification and NotSpecification主要为了组合
    **public class AndSpecification<T> : CompositeSpecification<T>
    {
        private  readonly ISpecification<T> _lefSpecification;
        private readonly ISpecification<T> _rightSpecification;

        public AndSpecification(ISpecification<T> left, ISpecificat
        {
            this._lefSpecification = left;
            this._rightSpecification = right;
        }

        public override bool IsSatisfiedBy(T candidate)
        {
            return this._lefSpecification.IsSatisfiedBy(candidate)
                && this._rightSpecification.IsSatisfiedBy(candidate
        }
    }

**public class OrSpecification<T> : CompositeSpecification<T>**
    {
        private readonly ISpecification<T> _leftSpecification;
        private readonly ISpecification<T> _rightSpecification;

        public OrSpecification(ISpecification<T> left, ISpecificati
        {
            this._leftSpecification = left;
            this._rightSpecification = right;
        }

        public override bool IsSatisfiedBy(T candidate)
        {
            return _leftSpecification.IsSatisfiedBy(candidate)
                || _rightSpecification.IsSatisfiedBy(candidate);
        }
    }

 **public class NotSpecification<T> : CompositeSpecification<T>**
    {
        private readonly ISpecification<T> _specification;

        public NotSpecification(ISpecification<T> specification)
        {
            this._specification = specification;
        }
        public override bool IsSatisfiedBy(T candidate)
        {
            return !_specification.IsSatisfiedBy(candidate);
        }
    }
```

接下来我们可以定义具体的规约模式，如果IdEqualSpecification、NameEqualSpecification规约等。下面就看下引入规约模式后，是如何解决上面仓储接口设计所存在的问题的。

```
// 引入规约模式，IProductRespository接口的定义
    public interface IProductRespository
    {
        Product GetBySpecification(ISpecification<Product> spec);
        IEnumerable<Product> FindBySpecification(ISpecification<Pro
    }

    public class IdEqualSpecification : CompositeSpecification<Prod
    {
        private readonly Guid _id;
        public IdEqualSpecification(Guid id)
        {
            _id = id;
        }

        public override bool IsSatisfiedBy(Product candidate)
        {
            return candidate.Id.Equals(_id);
        }
    }

    public class NameEqualSpecification : CompositeSpecification<Pr
    {
        private readonly string _name;

        public NameEqualSpecification(string name)
        {
            _name = name;
        }

        public override bool IsSatisfiedBy(Product candidate)
        {
            return candidate.Name.Equals(_name);
        }
    }

    public class NewProductsSpecification : CompositeSpecification<
    {
        public override bool IsSatisfiedBy(Product candidate)
        {
            return candidate.IsNew == true;
        }
    }
```

通过引入规约后，Product仓储中所有特定用途的操作都删除了，取而代之的是2个非常简洁的方法。规约模式解耦了仓储操作和筛选条件，如果业务扩展，我们可以定制我们的Specification，并将其注入到仓储即可。仓储的接口和实现无需任何修改。

下面通过一个具体的演示例子来看下传统规约模式的应用。具体的场景是这样的：我们想筛选一批int数组中的偶数和大于0的数字出来。因为这里涉及2个筛选条件，一个是偶数，一个是大于0的数，这样我们就可以通过定义偶数规约和正数规约。具体的实现如下所示：

```
// 具体规约，偶数规约
    public class EvenSpecification : CompositeSpecification<int>
    {
        public override bool IsSatisfiedBy(int candidate)
        {
            return candidate % 2 == 0;
        }
    }

    // 具体的规约，正数规约
    public class PlusSpecification  : CompositeSpecification<int>
    {
        public override bool IsSatisfiedBy(int candidate)
        {
            return candidate > 0;
        }
    }
```

接下来通过And操作和将2中规约组合起来形成新的规约。具体的测试代码如下所示：

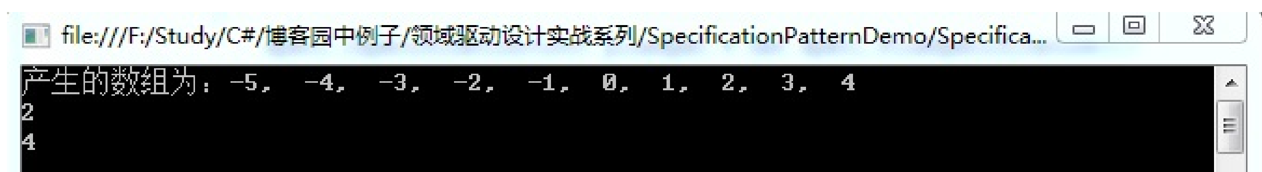
```
using spec1 =SpecificationPatternDemo.Specification;

class Program
    {
        static void Main(string[] args)
        {
            Demo1();
            Console.Read();
        }

        public static void Demo1()
        {
            var items = Enumerable.Range(-5, 10);
            Console.WriteLine("产生的数组为：{0}", string.Join(",  ",
            spec1.ISpecification<int> evenSpec = new spec1.EvenSpec
            // 获得一个组合规约
            var compositeSpecification = GetCompositeSpecification(
            // 类似Where(it=>it%2==0 && it > 0)
            // 前者是把两个条件合并写死成一个条件，而后者是将其组合成一个新条
            foreach (var item in items.Where(it=>compositeSpecifica
            {
                // 输出既是正数又是偶数的数
                Console.WriteLine(item);
            }
        }

        private static spec1.ISpecification<int> GetCompositeSpecif
        {
            spec1.ISpecification<int> plusSpec = new spec1.PlusSpec
            return spec.And(plusSpec);
        }
}
```

具体的运行结果如下图所示：



上面我们已经介绍完规约模式的实现，并且通过对比的方式来介绍引入规约模式所解决之前的问题。但是传统规约模式的实现显得非常臃肿。因为你想实现一个新的规约，你需要新增一个新的Specification类，这样下来，我们的项目中必然会堆积大量的Specification类。有些规约可能只只使用了一次。这就好比.NET里的委托方法一样，为了解决类似的问题，.NET引入了匿名方法和lamada表达式。同样，我们借助C#的特性也可以使得传统规约模式的实现更轻量。下面就具体看下规约模式的轻量实现是如何去实现的。

五、规约模式的轻量实现

从上面可以看出，规约模式的关键在于IsSatisifiedBy函数，该函数用于校验某个对象是否满足该规约所表示的条件，IsSatisifiedBy函数的返回类型为bool类型，这样我们完全可以让一个ISpecification只具有IsSatisifiedBy函数。然后该函数返回一个委托调用结果。至于原本ISpecification中的And、Or和Not方法，我们将它们提起成扩展方法。经过上面的分析，轻量化后的规约模式实现也就出来了。具体实现如下所示：

```
public interface ISpecification<in T>
   {
       bool IsSatisfiedBy(T candidate);
   }

   public class Specification<T> : ISpecification<T>
   {
       private readonly Func<T, bool> _isSatisfiedBy;

       public Specification(Func<T, bool> isSatisfiedBy)
       {
           this._isSatisfiedBy = isSatisfiedBy;
       }

       public bool IsSatisfiedBy(T candidate)
       {
           return _isSatisfiedBy(candidate);
       }
   }

   public static class SpecificationExtensions
   {
       public static ISpecification<T> And<T>(this ISpecification<
       {
           return new Specification<T>(candidate => left.IsSatisfi
       }

       public static ISpecification<T> Or<T>(this ISpecification<T
       {
           return new Specification<T>(candidate => left.IsSatisfi
       }

       public static ISpecification<T> Not<T>(this ISpecification<
       {
           return new Specification<T>(candidate => !one.IsSatisfi
       }
   }
```

使用扩展方法的好处在于，如果我们要加一个逻辑运行，如异或，那么就不需要修改接口了。修改接口是一个不推荐的的事情。因为接口修改会破坏之前已经发布的接口实现。因此，一旦接口发布之后，它就不能被修改了。这意味着，我们在定义接口时应该仔细推敲，做到接口的职责应该尤其单一。

轻量的实现使得使用Specification对象容易多了，我们不需要为每段逻辑创建一个独立的Specification类。下面具体看下规约模式的轻量实现的使用示例：

```
using SpecificationPatternDemo.Specification_2;
using spec2 = SpecificationPatternDemo.Specification_2;
class Program
    {
        static void Main(string[] args)
        {
            Demo2();
            Console.Read();
        }

         public static void Demo2()
        {
            var items = Enumerable.Range(-5, 10);
            Console.WriteLine("产生的数组为：{0}", string.Join(",  ",
            spec2.ISpecification<int> evenSpec = new spec2.Specific

            var compositeSpec = GetCompositeSpecification2(evenSpec

            foreach (var i in items.Where(it => compositeSpec.IsSat
            {
                Console.WriteLine(i);
            }
        }

        private static spec2.ISpecification<int> GetCompositeSpecif
        {
            spec2.ISpecification<int> plusSpec = new spec2.Specific
            return spec.And(plusSpec);
        }
}
```

从上面的例子可以看出，此时并不需要定义单独的Specification对象了，只需要用委托来代替即可。其运行结果与上面传统实现一样。其实，还可以更简单，我们可以直接使用一个委托，而不不需要定义ISpecification接口和其Specification实现。其实现方式如下所示：

```
// 更轻量的实现
    public static  class SpecExtensions
    {
        public static Func<T, bool> And<T>(this Func<T, bool> left,
        {
            return candidate => left(candidate) && right(candidate)
        }

        public static Func<T, bool> Or<T>(this Func<T, bool> left,
        {
            return candidate => left(candidate) || right(candidate)
        }

        public static Func<T, bool> Not<T>(this Func<T, bool> one)
        {
            return candidate => !one(candidate);
        }
    }
```

上面的实现，我们就只需要一个扩展方式就可以了，其使用示例代码如下所示：

```
class Program
    {
        static void Main(string[] args)
        {
            Demo3();
            Console.Read();
        }

        public static void Demo3()
        {
            var items = Enumerable.Range(-5, 10);
            Console.WriteLine("产生的数组为：{0}", string.Join(",  ",

           Func<int, bool> evenSpec = it => it % 2 == 0;

            var compositeSpec = GetCompositeSpec(evenSpec);

            foreach (var i in items.Where(it => compositeSpec(it)))
            {
                Console.WriteLine(i);
            }
        }

        private static Func<int, bool> GetCompositeSpec(Func<int, b
        {
            return spec.And(it => it > 0);
        }
    }
```

六、规约模式的轻量实现的完善——对Linq查询支持

上面轻量级的Specification模式抛弃了具体的Specification类型，而是使用一个委托对象关键的IsSatisfiedBy方法。其优势在于使用简单，但是该实现不能支持Linq查询或表达式的场景。因为EF中的DbContext.Dbset集合的进行where筛选参数只能是表达式树。所以我们不能用委托对象来判断逻辑，取而代之的使用表达式树。对于表达式树的构造主要由参数和主体构造，所以针对于Not方法可以如下的方式来实现：

```
 public static class SpecExprExtensions
    {
        public static Expression<Func<T, bool>> Not<T>(this Express
        {
            var candidateExpr = one.Parameters[0];
            var body = Expression.Not(one.Body);

            return Expression.Lambda<Func<T, bool>>(body, candidate
        }
}
```

对于Not方法，我们只要获取它的参数表达式，再将它的Body外包一个Not表达式，便可以此构造一个新的表达式了。但And和Or方法实现不能像Not一样简单处理：

```
// 不能这么处理
public static Expression<Func<T, bool>> And<T>(
    this Expression<Func<T, bool>> one, Expression<Func<T, bool>> a
{
    var candidateExpr = one.Parameters[0];
    var body = Expression.And(one.Body, another.Body);
    return Expression.Lambda<Func<T, bool>>(body, candidateExpr);
}
```

因为one和another两个表达式虽然都是同样的形式（Expression<Func<T, bool>>），但是它们的“参数”不是同一个对象。即one.Body和another.Body并没有公用一个ParameterExpression实例，于是无论采用哪个表达式的参数，在Expression.Lambda方法调用的时候，都会出现body中的某个参数对象并没有出现在参数列表中的错误。

既然参数不一致，所以要实现And和Or方法，必须统一两个表达式树的参数。为了达到这个目标，我们可以利用ExpressionVisitor.aspx)类来实现。这里定义一个派生于ExpressionVisitor.aspx)的类。具体实现如下：

```
internal class ParameterReplacer : ExpressionVisitor
    {
        public ParameterReplacer(ParameterExpression paramExpr)
        {
            this.ParameterExpression = paramExpr;
        }

        public ParameterExpression ParameterExpression { get; priva

        public Expression Replace(Expression expr)
        {
            return this.Visit(expr);
        }

        protected override Expression VisitParameter(ParameterExpre
        {
            return this.ParameterExpression;
        }
    }
```

Expressionvistor可以用于求值、变形等各种操作。它提供了遍历表达式树的标准方式，如果你直接继承这个类并调用Visit方法（如上面Replace方法的实现一样），那么最终返回的结果便是传入的Expresssion参数本身。但是，如果你覆盖任意一个方法，返回了与传入时不同的对象，那么最终的结果就是一个新的Expression对象。就如上面VisitParameter方法实现一样。它直接返回我们定义的ParameterExpression对象。

通过上面分析，ParameterExpression类的作用是将一个表达式里的所有ParameterExpression替换成我们指定的新对象，这样就可以解决之前参数不一致的情况。所以我们And和Or方法的实现就是将两个表达式树参数替换成我们首先定义好的参数表达式。具体的实现方式如下所示：

```
public static class SpecExprExtensions
    {
        public static Expression<Func<T, bool>> Not<T>(this Express
        {
            var candidateExpr = one.Parameters[0];
            var body = Expression.Not(one.Body);

            return Expression.Lambda<Func<T, bool>>(body, candidate
        }

        public static Expression<Func<T, bool>> And<T>(this Express
            Expression<Func<T, bool>> another)
        {
            // 首先定义好一个ParameterExpression
            var candidateExpr = Expression.Parameter(typeof (T), "c
            var parameterReplacer = new ParameterReplacer(candidate

            // 将表达式树的参数统一替换成我们定义好的candidateExpr
            var left = parameterReplacer.Replace(one.Body);
            var right = parameterReplacer.Replace(another.Body);

            var body = Expression.And(left, right);

            return Expression.Lambda<Func<T, bool>>(body, candidate
        }

        public static Expression<Func<T, bool>> Or<T>(
            this Expression<Func<T, bool>> one, Expression<Func<T,
        {
            var candidateExpr = Expression.Parameter(typeof (T), "c
            var parameterReplacer = new ParameterReplacer(candidate

            var left = parameterReplacer.Replace(one.Body);
            var right = parameterReplacer.Replace(another.Body);
            var body = Expression.Or(left, right);

            return Expression.Lambda<Func<T, bool>>(body, candidate
        }
    }
```

到此，我们就完成了规约模式对Linq支持的轻量实现了。下面让我们看看上面轻量实现是如何调用的呢？具体调用代码如下：

```
class Program
    {
        private static void Main(string[] args)
        {
            Demo1();
            Console.Read();
        }

        public static void Demo1()
        {
            var items = Enumerable.Range(-5, 10);
            Console.WriteLine("产生的数组为：{0}", string.Join(",  ",

            Expression<Func<int, bool>> f = i => i % 2 != 0;
            f = f.Not().And(i => i > 0);

            // 通过AsQueryable成IQueryable<int>,因为IQueryable<T>的Wh
            foreach (var i in items.AsQueryable().Where(f))
            {
                Console.WriteLine(i);
            }
        }
    }
```

其运行结果与前面的例子中的运行结果一样，一样成功返回了即是偶数又是正数的集合。

七、总结

到这里，规约模式的实现就结束了，后期将会在网上书店的案例中引入规约模式，dax.net的Byteart Retail案例中规约模式的实现即包括了传统实现，也包括了对Linq支持的轻量实现。开始我认为传统实现是多余的，因为你已经有了规约模式的轻量实现了，何必又有传统实现呢？这不是包括两种实现吗？后面仔细想想，这样设计也有其存在的道理，因为对于一些逻辑复杂的规约实现，我们可以新建一个具体的规约类，但对于一些简单和仅使用一次的规约逻辑，就可以直接用表达式树来代替，就不需要单独为该段逻辑单独新建一个具体的规约类。这样的实现就如同，有了匿名方法和Lambda表达式，是不是委托就可以不需要了。显然不是的，所以我在我的网上书店案例中也将会引入这两种实现，让用户可以灵活选择这两种方式。在下一专题，我继续介绍一个前期准备的内容，即工作单元模式(Unit Of Work，即UOW)。

本专题的所有源码下载：SpecificationPatternDemo.zip

[.NET领域驱动设计实战系列]专题四：前期准备之工作单元模式(Unit Of Work)

一、前言

在前一专题中介绍了规约模式的实现，然后在仓储实现中，经常会涉及工作单元模式的实现。然而，在我的网上书店案例中也将引入工作单元模式，所以本专题将详细介绍下该模式，为后面案例的实现做一个铺垫。

二、什么是工作单元模式（Unit Of Work）

工作单元模式：用来维护一个已经被业务事务修改（包括添加、修改或更新）的业务对象列表。工作单元模式复制协调这些修改的持久化工作以及所有标记的并发问题。采用工作单元模式带来的好处是能够保证数据的完整性。如果在持久化一系列业务对象的过程中出现问题，则将所有的修改回滚，以保证数据始终处于有效状态。

简单来说，工作单元模式就是把业务对象的持久化由工作单元实现类进行统一管理。而不想之前那样，分布在每个具体的仓储类中，这样就可以达到一系列业务对象的统一管理，不至于在每个业务对象中出现统一的提交和回滚业务逻辑，实现代码最大化重用。

三、工作单元模式的实现

从工作单元模式的定义可以看出，工作单元需要保存被业务事务修改的业务对象列表，则必须定义3个集合，分别是添加、修改和更新集合，然而如果使用EF的话，则不需要了，因为DbContext.DbSet<T>就可以代替这三个集合。这里为了演示，我们并没有引入EF，所以我们实现中定义了3个集合来保存被修改的业务对象。既然要在工作单元类中进行统一管理，则我们就需要在工作单元类中定义一个Commit方法，该方法的实现就是去遍历这三个集合的对象，对它们进行统一提交，如果其中一个失败，则进行数据回滚。根据面向接口编程原则，我们需要定义一个工作单元接口，即IUnitOfWork接口。经过上面的分析，再结合下面具体的实现来理解，工作单元模式的实现也就更加清晰了，下面让我们一起去实现下工作单元模式。

第一步：我们需要定义我们的领域层。

这里以银行账号之间转账业务作为背景，自然涉及到银行账号业务对象。并且领域层同时包括仓储接口的定义和领域服务。

领域服务指的是：如果有些方法涉及多个实体或聚合的交互，此时就应该把这段逻辑放到领域服务中，领域服务只有方法没有属性，也就是没有状态的。在银行转账业务中，账户之间的转账操作就适合作为领域服务来提供，因为转账操作涉及多个

聚合间的交互，需要从一个账户扣钱和另一个账户加钱。所以在领域层还需要定义账户转账服务。经过这样的分析，则领域层的实现如下所示：

```
    // 账号仓储接口
    public interface IAccountRepository
    {
        void Save(Account account);
        void Add(Account account);
        void Remove(Account account);
    }

    // 账号实体类
    public class Account : IAggregateRoot
    {
        public decimal Balance { get; set; }

        public System.Guid Id { get; set; }

        public Account()
        {
            Id = Guid.NewGuid();
        }
    }

    // 账号转账领域服务类
    public class AccountService
    {
        private readonly IAccountRepository _productRepository;
        private readonly IUnitOfWork _unitOfWork;

        public AccountService(IAccountRepository productRepository,
        {
            _productRepository = productRepository;
            _unitOfWork = unitOfWork;
        }

        public void Transfer(Account from, Account to, decimal amou
        {
            if (from.Balance >= amount)
            {
                from.Balance -= amount;
                to.Balance += amount;

                _productRepository.Save(from);
                _productRepository.Save(to);
                _unitOfWork.Commit();
            }
        }
    }
```

第二步：构建基础设施层

我们一般把工作单元模式的实现放在基础设施层，因为工作单元模式属于一种技术支持。根据上面工作单元模式的分析，我们首先定义IUnitOfWork接口，接着定义它的实现。因为这里没有引入EF，所以具体的实体的持久化还是调用具体的仓储来实现持久化的，所以还需要定义一个IUnitOfWorkRepository接口。则基础设施层的实现如下所示：

```
// 工作单元接口
    public interface IUnitOfWork
    {
        void RegisterAmended(IAggregateRoot entity, IUnitOfWorkRepo
        void RegisterNew(IAggregateRoot entity, IUnitOfWorkReposito
        void RegisterRemoved(IAggregateRoot entity, IUnitOfWorkRepo
        void Commit();
    }

    // 工作单元实现
    public class UnitOfWork : IUnitOfWork
    {
        // 引入了EF就不需要额外定义三个列表了，因为EF框架中包含的DbContext.
        // 然而在ByteartRetail案例中，也定义这3个列表，但其没有被真真使用到
        private readonly Dictionary<IAggregateRoot, IUnitOfWorkRepo
        private readonly Dictionary<IAggregateRoot, IUnitOfWorkRepo
        private readonly Dictionary<IAggregateRoot, IUnitOfWorkRepo

        public UnitOfWork()
        {
            _addedEntities = new Dictionary<IAggregateRoot, IUnitOf
            _changedEntities = new Dictionary<IAggregateRoot, IUnit
            _deletedEntities = new Dictionary<IAggregateRoot, IUnit
        }

        // 将业务对象实体添加到内部列表中，真正完成实体持久化操作的还是由具体
        public void RegisterAmended(IAggregateRoot entity, IUnitOfW
        {
            if (!_changedEntities.ContainsKey(entity))
            {
                _changedEntities.Add(entity, unitofWorkRepository);
            }
        }

        public void RegisterNew(IAggregateRoot entity, IUnitOfWorkR
        {
            if (!_addedEntities.ContainsKey(entity))
            {
                _addedEntities.Add(entity, unitofWorkRepository);
            };
        }

        public void RegisterRemoved(IAggregateRoot entity, IUnitOfW
        {
```

```
            if (!_deletedEntities.ContainsKey(entity))
            {
                _deletedEntities.Add(entity, unitofWorkRepository);
            }
        }

        protected void ClearRegisterations()
        {
            _addedEntities.Clear();
            _changedEntities.Clear();
            _deletedEntities.Clear();
        }

        // 对内部列表进行统一提交
        // 引入EF后，提交的实现有点不同，它具体的持久化只需要调用DbContext.
        // 则具体的仓储接口不需要实现IUnitOfWorkRepository接口，则自然不存
        public void Commit()
        {
            // 事务范围
            using (var scope = new TransactionScope())
            {
                // 分别调用具体的仓储对象的持久化逻辑来对业务对象进行持久化
                foreach (var entity in this._addedEntities.Keys)
                {
                    this._addedEntities[entity].PersistCreationOf(e
                }

                foreach (var entity in this._changedEntities.Keys)
                {
                    this._changedEntities[entity].PersistUpdateOf(e
                }

                foreach (var entity in this._deletedEntities.Keys)
                {
                    this._deletedEntities[entity].PersistDeletionOf
                }

                scope.Complete();
            }

             // 清楚内存中对象
            ClearRegisterations();
        }
    }

    public interface IUnitOfWorkRepository
    {
        Hashtable AccountList { get; }
        void PersistCreationOf(IAggregateRoot entity);
        void PersistUpdateOf(IAggregateRoot entity);
        void PersistDeletionOf(IAggregateRoot entity);
    }
```

```
    public interface IAggregateRoot
    {
        Guid Id { get; }
    }
```

第三步：实现仓储层。

仓储的实现在之前也说过，它其实可以放在基础设施层里，但一般总将其放在一个单独层进行实现。所以这里也就放在一个单独层进行实现。这里仓储实现只有一个类，即对IAccountRepository接口的实现。具体的实现代码如下所示：

```
public class AccountRepository : IAccountRepository, IUnitOfWorkRe
    {
        private readonly IUnitOfWork _unitOfWork;

        public AccountRepository(IUnitOfWork unitOfWork)
        {
            _unitOfWork = unitOfWork;
            AccountList = new Hashtable();
        }

        public Hashtable AccountList { get; set; }

        #region IAccountRepository Members

        public void Save(Account account)
        {
            _unitOfWork.RegisterAmended(account, this);
        }

        public void Add(Account account)
        {
            _unitOfWork.RegisterNew(account, this);
        }

        public void Remove(Account account)
        {
            _unitOfWork.RegisterRemoved(account, this);
        }

        #endregion

        #region IUnitOfWorkRepository Members

        public void PersistUpdateOf(IAggregateRoot entity)
        {
            // ADO.net code to update the entity...
            // 这里为了演示，只它持久化到内存中
            if (AccountList.ContainsKey(entity.Id))
            {
```

```
                AccountList[entity.Id] = entity;
            }
        }

        public void PersistCreationOf(IAggregateRoot entity)
        {
            // ADO.net code to Add the entity...
            // 这里为了演示，只它持久化到内存中
            AccountList.Add(entity.Id, entity);
        }

        public void PersistDeletionOf(IAggregateRoot entity)
        {
            // ADO.net code to delete the entity...
            // 这里为了演示，只它持久化到内存中
            if (AccountList.ContainsKey(entity.Id))
            {
                AccountList.Remove(entity.Id);
            }
        }
        #endregion
    }
```

这样，就完成了工作单元模式模式的实现和应用了，工作单元模式的实现存在于基础设施层，其他层的构建主要为了演示，下面通过一个测试项目来进行测试下工作单元的使用。具体项目的引入将会在下一个专题中介绍，下一个专题将引入工作单元模式和规约模式的实现。具体的测试项目的代码如下所示：

```
[TestClass]
    public class AccountRepositoryTests
    {
        [TestMethod]
        public void AccountRepository_Delegates_Changes_To_The_Unit
        {
            var accountToBeAmended = new Account();
            var accountToBeRemoved = new Account();
            var accountToBeAdded = new Account();

            // 需要引入Moq Mock框架
            var unitOfWorkMockery = new Mock<IUnitOfWork>();

            var accountRepository = new AccountRepository(unitOfWor

            unitOfWorkMockery.Setup(uow => uow.RegisterAmended(acco
            unitOfWorkMockery.Setup(uow => uow.RegisterNew(accountT
            unitOfWorkMockery.Setup(uow => uow.RegisterRemoved(acco

            accountRepository.Add(accountToBeAdded);
            accountRepository.Save(accountToBeAmended);
            accountRepository.Remove(accountToBeRemoved);

            unitOfWorkMockery.VerifyAll();
        }
    }
```

四、总结

到这里，本专题的内容就介绍完了，上一专题和这一专题都是一个前期准备的专题，主要是为网上书店案例引入这2个模式的实现做一个铺垫，为了让大家对知识点分开吸收，然后再通过后面一专题的应用来加深理解这2个模式的应用。

本专题的所有源码下载：UnitOfWorkDemo.zip

[.NET领域驱动设计实战系列]专题五：网上书店规约模式、工作单元模式的引入以及购物车的实现

一、前言

在前面2篇博文中，我分别介绍了规约模式和工作单元模式，有了前面2篇博文的铺垫之后，下面就具体看看如何把这两种模式引入到之前的网上书店案例里。

二、规约模式的引入

在第三专题我们已经详细介绍了什么是规约模式，没看过的朋友首先去了解下。下面让我们一起看看如何在网上书店案例中引入规约模式。在网上书店案例中规约模式的实现兼容了2种模式的实现，兼容了传统和轻量的实现，包括传统模式的实现，主要是为了实现一些共有规约的重用，不然的话可能就要重复写这些表达式。下面让我们具体看看在该项目中的实现。

首先是ISpecification接口的定义以及其抽象类的实现

```
public interface ISpecification<T>
    {
        bool IsSatisfiedBy(T candidate);
        Expression<Func<T, bool>> Expression { get; }
    }

 public abstract class Specification<T> : ISpecification<T>
    {
        public static Specification<T> Eval(Expression<Func<T, bool
        {
            return new ExpressionSpecification<T>(expression);
        }

        #region ISpecification<T> Members
        public bool IsSatisfiedBy(T candidate)
        {
            return this.Expression.Compile()(candidate);
        }

        public abstract Expression<Func<T, bool>> Expression { get;
        #endregion
    }
```

上面的实现稍微对传统规约模式进行了一点修改，添加 Expression属性来获得规约的表达式树。另外，在该案例中还定义了一个包装表达式树的规约类和没有任何条件的规约类AnySpecification。其具体实现如下：

```
public sealed class ExpressionSpecification<T> : Specification<T>
    {
        private readonly Expression<Func<T, bool>> _expression;
        public ExpressionSpecification(Expression<Func<T, bool>> ex
        {
            this._expression = expression;
        }

        public override Expression<Func<T, bool>> Expression
        {
            get { return _expression; }
        }
    }

public sealed class AnySpecification<T> : Specification<T>
    {
        public override Expression<Func<T, bool>> Expression
        {
            get { return o => true; }
        }
    }
```

接下来就是轻量规约模式的实现了，该实现涉及2个类，一个是包含扩展方法的类和一个实现统一表达式树参数的类。具体实现代码如下所示：

```
public static class SpecExprExtensions
    {
        public static Expression<Func<T, bool>> Not<T>(this Express
        {
            var candidateExpr = one.Parameters[0];
            var body = Expression.Not(one.Body);

            return Expression.Lambda<Func<T, bool>>(body, candidate
        }

        public static Expression<Func<T, bool>> And<T>(this Express
            Expression<Func<T, bool>> another)
        {
            // 首先定义好一个ParameterExpression
            var candidateExpr = Expression.Parameter(typeof(T), "ca
            var parameterReplacer = new ParameterReplacer(candidate

            // 将表达式树的参数统一替换成我们定义好的candidateExpr
            var left = parameterReplacer.Replace(one.Body);
            var right = parameterReplacer.Replace(another.Body);
```

```
            var body = Expression.And(left, right);

            return Expression.Lambda<Func<T, bool>>(body, candidate
        }

        public static Expression<Func<T, bool>> Or<T>(
            this Expression<Func<T, bool>> one, Expression<Func<T,
        {
            var candidateExpr = Expression.Parameter(typeof(T), "ca
            var parameterReplacer = new ParameterReplacer(candidate

            var left = parameterReplacer.Replace(one.Body);
            var right = parameterReplacer.Replace(another.Body);
            var body = Expression.Or(left, right);

            return Expression.Lambda<Func<T, bool>>(body, candidate
        }
    }

public class ParameterReplacer : ExpressionVisitor
    {
        public ParameterReplacer(ParameterExpression paramExpr)
        {
            this.ParameterExpression = paramExpr;
        }

        public ParameterExpression ParameterExpression { get; priva

        public Expression Replace(Expression expr)
        {
            return this.Visit(expr);
        }

        protected override Expression VisitParameter(ParameterExpre
        {
            return this.ParameterExpression;
        }
    }
```

这样，规约模式在案例中的实现就完成了，下面具体介绍下工作单元模式是如何在该项目中实现的。

三、工作单元模式的引入

工作单元模式，主要是为了保证数据的一致性，一些涉及到多个实体的操作我们希望它们一起被提交，从而保证数据的正确性和一致性。例如，在该项目，用户成功注册的同时需要为用户创建一个购物车对象，这里就涉及到2个实体，一个是用户实体，一个是购物车实体，所以此时必须保证这个操作必须作为一个操作被提交，

这样就可以保证要么一起提交成功，要么都失败，不存在其中一个被提交成功的情况，否则就会出现数据不正确的情况，上一专题的转账业务也是这个道理，只是转账业务涉及的是2个相同的实体，都是账户实体。

从上面描述可以发现，要保证数据的一致性，必须要有一个类统一管理提交操作，而不能由其仓储实现来提交数据改变。根据上一专题我们可以知道，首先需要定义一个工作单元接口IUnitOfWork，工作单元接口的定义通常放在基础设施层，其定义代码如下所示：

在该项目中，对工作单元接口的方法进行了一个分离，把其方法分别定义在2个接口中，工作单元接口中仅仅定义了一个Commit方法，RegisterNew, RegisterModified和RegisterDelete方法定义在IRepositoryContext接口中。当然我觉得也可以把这4个操作都定义在IUnitOfWork接口中。这里只是遵循dax.net案例中设计来实现的。然后EntityFrameworkRepositoryContext来实现这4个操作。工作单元模式在项目中的实现代码如下所示：

```
// 仓储上下文接口
    // 这里把传统的IUnitOfWork接口中方法分别在2个接口定义：一个是IUnitOfWo
    public interface IRepositoryContext : IUnitOfWork
    {
        // 用来标识仓储上下文
        Guid Id { get; }

        void RegisterNew<TAggregateRoot>(TAggregateRoot entity)
            where TAggregateRoot : class, IAggregateRoot;

        void RegisterModified<TAggregateRoot>(TAggregateRoot entity
            where TAggregateRoot : class, IAggregateRoot;

        void RegisterDeleted<TAggregateRoot>(TAggregateRoot entity
            where TAggregateRoot : class, IAggregateRoot;
    }

 public interface IEntityFrameworkRepositoryContext : IRepositoryCo
    {
        #region Properties
        OnlineStoreDbContext DbContex { get; }
        #endregion
    }

// IEntityFrameworkRepositoryContext接口的实现
    public class EntityFrameworkRepositoryContext : IEntityFramewor
    {
        // ThreadLocal代表线程本地存储，主要相当于一个静态变量
        // 但静态变量在多线程访问时需要显式使用线程同步技术。
        // 使用ThreadLocal变量，每个线程都会一个拷贝，从而避免了线程同步带来

        private readonly ThreadLocal<OnlineStoreDbContext> _localCt
        public OnlineStoreDbContext DbContex
        {
            get { return _localCtx.Value; }
```

```
        }

        private readonly Guid _id = Guid.NewGuid();

        #region IRepositoryContext Members
        public Guid Id
        {
            get { return _id; }
        }

        public void RegisterNew<TAggregateRoot>(TAggregateRoot enti
        {
            _localCtx.Value.Set<TAggregateRoot>().Add(entity);
        }

        public void RegisterModified<TAggregateRoot>(TAggregateRoot
        {
            _localCtx.Value.Entry<TAggregateRoot>(entity).State = E
        }

        public void RegisterDeleted<TAggregateRoot>(TAggregateRoot 
        {
            _localCtx.Value.Set<TAggregateRoot>().Remove(entity);
        }

        #endregion

        #region IUnitOfWork Members
        public void Commit()
        {
            var validationError = _localCtx.Value.GetValidationErro
            _localCtx.Value.SaveChanges();
        }
        #endregion
    }
```

到此，工作单元模式的引入也就完成了，接下面，让我们继续完成网上书店案例。

四、购物车的实现

在前一个版本中，只是实现了商品的展示和详细信息等功能。在网上商店中，都有购物车这个功能，作为一个完整的案例，该案例也不能少了这个功能。在实现购物车之前，我们首先理清下业务逻辑：访问者看到商品，然后点击商品下的加入购物车按钮，把商品加入购物车。

在上面的业务逻辑中，涉及了下面几个更细的业务逻辑：

- 如果用户没注册时，访客点击加入购物车按钮应跳转到注册界面，这样就涉及到用户注册功能的实现

- 用户注册成功后需要同时为用户创建一个购物车实例与该用户进行绑定，之后用户就可以把商品加入自己的购物车
- 加入购物车之后，用户可以查看购物车中的商品，同时也应该可以进行更新和移除操作。

通过上面的描述，大家应该自然明白了我们接下来需要哪些功能了吧，下面我们来理理：

1. 用户注册功能，涉及用户注册页面。自然就涉及用户注册服务和用户仓储的实现
2. 注册成功同时创建购物车实例。自然涉及创建购物车服务方法和购物车仓储的实现
3. 加入购物车成功后，可以查看购物车中的商品、更新和移除操作，自然涉及到购物车页面的实现。这里自然涉及到获得购物车和更新商品数量和删除购物项的服务方法。

理清了思路之后，接下来就应该去实现功能了，首先应该实现自然就是用户注册模块。实现功能模块都从底向上来实现，首先应该先定义用户聚合根，接着实现用户仓储和用户服务，最后实现控制器和视图。下面是用户注册涉及的主要类的实现：

```
// 用户聚合根
    public class User : AggregateRoot
    {
        public string UserName { get; set; }
        public string Password { get; set; }

        public string Email { get; set; }

        public string PhoneNumber { get; set; }

        public bool IsDisabled { get; set; }

        public DateTime RegisteredDate { get; set; }

        public DateTime? LastLogonDate { get; set; }

        public string Contact { get; set; }
        //用户的联系地址
        public Address ContactAddress { get; set; }

        //用户的发货地址
        public Address DeliveryAddress { get; set; }

        public IEnumerable<Order> Orders
        {
            get
            {
                IEnumerable<Order> result = null;
                //DomainEvent.Publish<GetUserOrdersEvent>(new GetUs
                //    (e, ret, exc) =>
                //    {
```

```
                //        result = e.Orders;
                //    });
                return result;
            }
        }

        public override string ToString()
        {
            return this.UserName;
        }

        #region Public Methods

        public void Disable()
        {
            this.IsDisabled = true;
        }

        public void Enable()
        {
            this.IsDisabled = false;
        }

        // 为当前用户创建购物篮。
        public ShoppingCart CreateShoppingCart()
        {
            var shoppingCart = new ShoppingCart { User = this };
            return shoppingCart;
        }
        #endregion
    }

public interface IUserRepository : IRepository<User>
    {
        bool CheckPassword(string userName, string password);
    }

 public class UserRepository : EntityFrameworkRepository<User>, IUs
    {
        public UserRepository(IRepositoryContext context)
            : base(context)
        {
        }

        public bool CheckPassword(string userName, string password)
        {
            Expression<Func<User, bool>> userNameExpression = u =>
            Expression<Func<User, bool>> passwordExpression = u =>

            return Exists(new ExpressionSpecification<User>(userNar
        }
    }
```

```
// 用户服务契约
    [ServiceContract(Namespace = "")]
    public interface IUserService
    {
        #region Methods

        [OperationContract]
        [FaultContract(typeof (FaultData))]
        IList<UserDto> CreateUsers(List<UserDto> userDtos);

        [OperationContract]
        [FaultContract(typeof(FaultData))]
        bool ValidateUser(string userName, string password);

        [OperationContract]
        [FaultContract(typeof(FaultData))]
        bool DisableUser(UserDto userDto);

        [OperationContract]
        [FaultContract(typeof(FaultData))]
        bool EnableUser(UserDto userDto);

        [OperationContract]
        [FaultContract(typeof(FaultData))]
        void DeleteUsers(UserDto userDto);

        [OperationContract]
        [FaultContract(typeof(FaultData))]
        IList<UserDto> UpdateUsers(List<UserDto> userDataObjects);

        [OperationContract]
        [FaultContract(typeof (FaultData))]
        UserDto GetUserByKey(Guid id);

        [OperationContract]
        [FaultContract(typeof(FaultData))]
        UserDto GetUserByEmail(string email);

        [OperationContract]
        [FaultContract(typeof(FaultData))]
        UserDto GetUserByName(string userName);

        #endregion
    }

 public class **UserServiceImp** :ApplicationService, IUserService
    {
        private readonly IUserRepository _userRepository;
        private readonly IShoppingCartRepository _shoppingCartRepos

        public UserServiceImp(IRepositoryContext repositoryContext,
            IUserRepository userRepository,
            IShoppingCartRepository shoppingCartRepository)
```

```
            : base(repositoryContext)
        {
            _userRepository = userRepository;
            _shoppingCartRepository = shoppingCartRepository;
        }

        public IList<UserDto> CreateUsers(List<UserDto> userDtos)
        {
            if (userDtos == null)
                throw new ArgumentNullException("userDtos");
            return PerformCreateObjects<List<UserDto>, UserDto, Use
                _userRepository,
                dto =>
                {
                    if (dto.RegisterDate == null)
                        dto.RegisterDate = DateTime.Now;
                },
                ar =>
                {
                    var shoppingCart = ar.CreateShoppingCart();
                    _shoppingCartRepository.Add(shoppingCart);
                });
        }

        public bool ValidateUser(string userName, string password)
        {
            if (string.IsNullOrEmpty(userName))
                throw new ArgumentNullException("userName");
            if (string.IsNullOrEmpty(password))
                throw new ArgumentNullException("password");

            return _userRepository.CheckPassword(userName, passworc
        }

        public bool DisableUser(UserDto userDto)
        {
            if(userDto == null)
                throw new ArgumentNullException("userDto");
            User user;
            if (!IsEmptyGuidString(userDto.Id))
                user = _userRepository.GetByKey(new Guid(userDto.Ic
            else if (!string.IsNullOrEmpty(userDto.UserName))
                user = _userRepository.GetByExpression(u=>u.UserNar
            else if (!string.IsNullOrEmpty(userDto.Email))
                user = _userRepository.GetByExpression(u => u.Emai
            else
                throw new ArgumentNullException("userDto", "Either
            user.Disable();
            _userRepository.Update(user);
            RepositorytContext.Commit();
            return user.IsDisabled;
        }
```

```
        public bool EnableUser(UserDto userDto)
        {
            if (userDto == null)
                throw new ArgumentNullException("userDto");
            User user;
            if (!IsEmptyGuidString(userDto.Id))
                user = _userRepository.GetByKey(new Guid(userDto.Ic
            else if (!string.IsNullOrEmpty(userDto.UserName))
                user = _userRepository.GetByExpression(u => u.UserN
            else if (!string.IsNullOrEmpty(userDto.Email))
                user = _userRepository.GetByExpression(u => u.Emai
            else
                throw new ArgumentNullException("userDto", "Either
            user.Enable();
            _userRepository.Update(user);
            RepositorytContext.Commit();
            return user.IsDisabled;
        }

        public IList<UserDto> UpdateUsers(List<UserDto> userDataObj
        {
            throw new NotImplementedException();
        }

        public void DeleteUsers(UserDto userDto)
        {
            throw new System.NotImplementedException();
        }

        public UserDto GetUserByKey(Guid id)
        {
            var user = _userRepository.GetByKey(id);
            var userDto = Mapper.Map<User, UserDto>(user);
            return userDto;
        }

        public UserDto GetUserByEmail(string email)
        {
            if(string.IsNullOrEmpty(email))
                throw new ArgumentException("email");
            var user = _userRepository.GetByExpression(u => u.Emai
            var userDto = Mapper.Map<User, UserDto>(user);
            return userDto;
        }

        public UserDto GetUserByName(string userName)
        {
            if (string.IsNullOrEmpty(userName))
                throw new ArgumentException("userName");
            var user = _userRepository.GetByExpression(u => u.UserN
            var userDto = Mapper.Map<User, UserDto>(user);
            return userDto;
        }
```

```
    }

    // UserService.svc, WCF服务
    [ServiceBehavior(InstanceContextMode = InstanceContextMode.PerS
    public class UserService : IUserService
    {
        private readonly IUserService _userServiceImp;

        public UserService()
        {
            _userServiceImp = ServiceLocator.Instance.GetService<IU
        }

        public IList<UserDto> CreateUsers(List<UserDto> userDtos)
        {
            try
            {
                return _userServiceImp.CreateUsers(userDtos);
            }
            catch (Exception ex)
            {
                throw new FaultException<FaultData>(FaultData.Creat
            }
        }

        public bool ValidateUser(string userName, string password)
        {
            try
            {
                return _userServiceImp.ValidateUser(userName, passw
            }
            catch (Exception ex)
            {
                throw new FaultException<FaultData>(FaultData.Creat
            }
        }

        public bool DisableUser(UserDto userDto)
        {
            try
            {
                return _userServiceImp.DisableUser(userDto);
            }
            catch (Exception ex)
            {
                throw new FaultException<FaultData>(FaultData.Creat
            }
        }

        public bool EnableUser(UserDto userDto)
        {
            try
            {
```

```
                return _userServiceImp.EnableUser(userDto);
            }
            catch (Exception ex)
            {
                throw new FaultException<FaultData>(FaultData.Creat
            }
        }

        public void DeleteUsers(UserDto userDto)
        {
            throw new NotImplementedException();
        }

        public IList<UserDto> UpdateUsers(List<UserDto> userDataObj
        {
            throw new NotImplementedException();
        }

        public **UserDto** GetUserByKey(Guid id)
        {
            try
            {
                return _userServiceImp.GetUserByKey(id);
            }
            catch (Exception ex)
            {
                throw new FaultException<FaultData>(FaultData.Creat
            }
        }

        public **UserDto** GetUserByEmail(string email)
        {
            try
            {
                return _userServiceImp.GetUserByEmail(email);
            }
            catch (Exception ex)
            {
                throw new FaultException<FaultData>(FaultData.Creat
            }
        }

        public **UserDto** GetUserByName(string userName)
        {
            try
            {
                return _userServiceImp.GetUserByName(userName);
            }
            catch (Exception ex)
            {
                throw new FaultException<FaultData>(FaultData.Creat
            }
        }
```

```
    }
```

从上面代码可以看出，这个版本应用服务的实现和前一个版本有一个很大的不同，首先应用接口的定义采用了数据传输对象,Data Transfer Object(DTO)。DTO对象作用是为了隔离Domain Model，让Domain Model的改动不会直接影响到UI，保证Domain Model的安全，不暴露业务逻辑。通过DTO我们实现了表现层与Model之间的解耦，表现层不引用Model，如果开发过程中我们的模型改变了，而界面没变，我们就只需要改Model而不需要去改表现层中的东西。关于DTO更详细的介绍可以参考：http://www.cnblogs.com/ego/archive/2009/05/13/1456363.html

其次，目前WCF服务并没有对WCF接口进行直接实现，而是通过引用WCF接口的实现类来完成的。之前的设计把WCF实现直接在WCF服务里面进行实现的。

用户注册成功之后，就可以用对应的账号进行登录，登录成功之后，就可以把对应的商品添加进购物车中，下面分别介绍登录功能和加入购物车功能的具体实现。

首先是登录功能的实现，其实现所涉及的代码如下所示：

```
 [Authorize]
    [HandleError]
    public class AccountController : Controller
    {
         // 登录按钮触发的操作
         [HttpPost]
        [AllowAnonymous]
        public ActionResult Login(LoginViewModel model, string retu
        {
            if (ModelState.IsValid)
            {
                if (Membership.ValidateUser(model.UserName, model.P
                {
                    FormsAuthentication.SetAuthCookie(model.UserNan
                    if (Url.IsLocalUrl(returnUrl) && returnUrl.Leng
                        && !returnUrl.StartsWith("//") && !returnUr
                    {
                        return Redirect(returnUrl);
                    }
                    else
                    {
                        return RedirectToAction("Index", "Home");
                    }
                }
                else
                {
                    ModelState.AddModelError("", "用户名或密码不正确！
                }
            }
            return View();
        }
    }
```

登录成功后，用户就可以把商品添加入购物车了，具体涉及的代码实现如下所示：

下面是HomeController中AddToCart操作的实现

```
public class HomeController : Controller
    {
        #region Protected Properties
        protected Guid UserId
        {
            get
            {
                if (Session["UserId"] != null)
                {
                    return (Guid) Session["UserId"];
                }
                else
                {
```

```
                    var id = new Guid(Membership.GetUser().Provider
                    Session["UserId"] = id;
                    return id;
                }
            }
        }
        #endregion

        public ActionResult Index()
        {
            return View();
        }

        [Authorize]
        public ActionResult AddToCart(string productId, string iter
        {
            using (var proxy = new OrderServiceClient())
            {
                int quantity = 0;
                if (!int.TryParse(items, out quantity))
                    quantity = 1;
                proxy.AddProductToCart(UserId, new Guid(productId),
                return RedirectToAction("ShoppingCart");
            }
        }

        [Authorize]
        public ActionResult ShoppingCart()
        {
            using (var proxy = new OrderServiceClient())
            {
                var model = proxy.GetShoppingCart(UserId);
                return View(model);
            }
        }

        [Authorize]
        public ActionResult UpdateShoppingCartItem(string shopping(
        {
            using (var proxy = new OrderServiceClient())
            {
                proxy.UpdateShoppingCartItem(new Guid(shoppingCartI
                return Json(null);
            }
        }

        [Authorize]
        public ActionResult DeleteShoppingCartItem(string shopping(
        {
            using (var proxy = new OrderServiceClient())
            {
                proxy.DeleteShoppingCartItem(new Guid(shoppingCartI
                return Json(null);
```

```
            }
        }
    }
```

从上面代码可以看出，HomeController中的AddToCart操作是通过调用应用层的OrderService来完成，而OrderService又是调用对应的仓储接口来完成数据的持久化的，即把对应的商品放进对应的用户的购物车对象中。关于应用层和仓储层的具体实现如下所示：

```
**// OrderService的实现**
public class OrderServiceImp : ApplicationService, IOrderService
    {
        #region Private Fileds
        private readonly IShoppingCartRepository _shoppingCartRepos
        private readonly IShoppingCartItemRepository _shoppingCartI
        private readonly IUserRepository _userRepository;
        private readonly IProductRepository _productRepository;

        #endregion

        #region Ctor
        public OrderServiceImp(IRepositoryContext context,
            IUserRepository userRepository,
            IShoppingCartRepository shoppingCartRepository,
            IProductRepository productRepository,
            IShoppingCartItemRepository shoppingCartItemRepository)
        {
            _userRepository = userRepository;
            _shoppingCartRepository = shoppingCartRepository;
            _productRepository = productRepository;
            _shoppingCartItemRepository = shoppingCartItemReposito
        }

        #endregion

        #region IOrderService Members

        public void AddProductToCart(Guid customerId, Guid productI
        {
            var user = _userRepository.GetByKey(customerId);

            var shoppingCart = _shoppingCartRepository.GetBySpecifi
            if (shoppingCart == null)
                throw new DomainException("用户{0}不存在购物车.", cus

            var product = _productRepository.GetByKey(productId);
            var shoppingCartItem = _shoppingCartItemRepository.Find
            if (shoppingCartItem == null)
            {
```

```
                shoppingCartItem = new ShoppingCartItem()
                {
                    Product = product,
                    ShoopingCart = shoppingCart,
                    Quantity = quantity
                };

                _shoppingCartItemRepository.Add(shoppingCartItem);
            }
            else
            {
                shoppingCartItem.UpdateQuantity(shoppingCartItem.Qu
                _shoppingCartItemRepository.Update(shoppingCartIter
            }

            RepositorytContext.Commit();
        }

        public ShoppingCartDto GetShoppingCart(Guid customerId)
        {
            var user = _userRepository.GetByKey(customerId);

            var shoppingCart = _shoppingCartRepository.GetBySpecifi
                new ExpressionSpecification<ShoppingCart>(s => s.Us
            if (shoppingCart == null)
                throw new DomainException("用户{0}不存在购物车.", cus

            var shoppingCartItems =
                _shoppingCartItemRepository.GetAll(
                    new ExpressionSpecification<ShoppingCartItem>(s

            var shoppingCartDto = Mapper.Map<ShoppingCart, Shopping
            shoppingCartDto.Items = new List<ShoppingCartItemDto>()
            if (shoppingCartItems != null && shoppingCartItems.Any(
            {
                shoppingCartItems
                    .ToList()
                    .ForEach(s => shoppingCartDto.Items.Add(Mapper.
                shoppingCartDto.Subtotal = shoppingCartDto.Items.Su
            }

            return shoppingCartDto;
        }

        public int GetShoppingCartItemCount(Guid userId)
        {
            var user = _userRepository.GetByKey(userId);
            var shoppingCart = _shoppingCartRepository.GetBySpecifi
            if(shoppingCart == null)
                throw new InvalidOperationException("没有可用的购物车
            var shoppingCartItems =
                _shoppingCartItemRepository.GetAll(new ExpressionSp
            return shoppingCartItems.Sum(s => s.Quantity);
```

```
        }

        public void UpdateShoppingCartItem(Guid shoppingCartItemId,
        {
            var shoppingCartItem = _shoppingCartItemRepository.GetB
            shoppingCartItem.UpdateQuantity(quantity);
            _shoppingCartItemRepository.Update(shoppingCartItem);
            RepositorytContext.Commit();
        }

        public void DeleteShoppingCartItem(Guid shoppingCartItemId)
        {
            var shoppingCartItem = _shoppingCartItemRepository.GetB
            _shoppingCartItemRepository.Remove(shoppingCartItem);
            RepositorytContext.Commit();
        }

        public OrderDto Checkout(Guid customerId)
        {
            throw new NotImplementedException();
        }
        #endregion

    }
**// 加入购物车所涉及仓储的实现** public class ShoppingCartRepository :
    {
        public ShoppingCartRepository(IRepositoryContext context) :
        {
        }
    }

public class ShoppingCartItemRepository : EntityFrameworkRepository
    {
        public ShoppingCartItemRepository(IRepositoryContext contex
            : base(context)
        {
        }

        #region IShoppingCartItemRepository Members
        public ShoppingCartItem FindItem(ShoppingCart shoppingCart,
        {
            return GetBySpecification(Specification<ShoppingCartIte
                (sci => sci.ShoopingCart.Id == shoppingCart.Id &&
                 sci.Product.Id == product.Id));
        }

        #endregion
    }
```

这样，也就完成购物车的实现，下面让我们要一起看看完善后网上书店的运行效果，

首先，如果没有登录的话，当用户点击商品上的购买按钮时，会自动跳转到登录界面，具体登录界面如下所示：



这里由于我演示的时候已经注册过一个账号了，这时候我就用注册好的账号进行登录，如果你没有账号的话，可以直接注册一个账号。登录成功之后，你就可以把对应商品添加进购物车，具体运行效果如下图所示：

并且，你还可以对购物车中商品进行操作，例如移除，数量更新操作等，如果此时更新Asp.net设计模式的数量为1的话，此时的运行效果如下图所示：

从上图可以发现，当我们更新商品的数量时，对应的总数量和总价也相应地进行了更新。当然你还可以对商品进行删除操作。这里就不一一贴图了。大家可以自行从github上下载源码运行看看。

五、总结

到这里，网上书店的购物车功能的实现就完成了，在接下来的系列中，我会继续完善这个DDD系列，在接下来的一个系列中将会对加入订单功能。

网上书店v0.2版Github下载地址：https://github.com/lizhi5753186/OnlineStore_Second

[.NET领域驱动设计实战系列]专题六：DDD实践案例：网上书店订单功能的实现

一、引言

上一专题已经为网上书店实现了购物车的功能了，在这一专题中，将继续对网上书店案例进行完善，本专题将对网上书店订单功能的实现进行介绍，现在废话不多说了，让我们来一起看看订单功能是如何实现的吧。

二、订单功能的实现思路

在网上购过物的朋友，对于订单功能的流程自然不陌生，这里我还是先来梳理下下订单的一个流程：

- 用户点击我的购物车，可以勾选对应的商品进行结算
- 在结算页面可以提交订单来创建一个订单
- 创建订单成功之后就是进行付款了。

一般购物网站下订单的流程分上面3步，由于在本案例中并没有对接第三方的支付平台，所以这里就没有上面第三步的过程了。即在网上书店案例中订单提交成功之后就是已付款状态。

从上面下订单流程我们就可以知道订单功能的实现思路：

- 用户点击购物车中购买商品按钮来进行下订单，此时触发控制器中的结算方法进行调用
- 结算方法通过调用OrderService服务中的结算方法
- 由于订单的创建涉及了3个实体操作，包括购物车实体，购物车项实体和订单实体。所以这里需要引入领域服务。因为创建订单这个操作涉及了多个实体，则这个业务逻辑放在每个实体中都不合适，因为并属于单独一个实体的逻辑。所以这里引入领域服务来实现这种涉及多个实体的操作。
- 则OrderService服务中的结算方法通过调用领域服务中的CreateOrder方法来完成订单创建的功能。
- 领域服务中可以调用对应的实体仓储来完成实体的持久化。这里需要注意的一点：因为领域服务涉及多个实体的持久化，则需要引入工作单元模式将这些实体的操作进行统一提交，要不都成功，要不都不成功。这也就是引入工作单元初衷。

上面的思路就是订单功能的实现思路。有了上面的思路之后，实现订单功能也一目了然了。下面让我们一起在网上书店案例中实现下看看。

三、网上书店订单功能的实现

这里我们按照上面的实现思路由下至上地去实现订单功能。

1. 首先我们需要订单仓储来完成订单实体的持久化。具体订单仓储接口和实现如下代码所示：

```
// 订单仓储接口
    public interface IOrderRepository : IRepository<Order>
    {
    }

 // 订单仓储的实现类
    public class OrderRepository : EntityFrameworkRepository<Order>
    {
        public OrderRepository(IRepositoryContext context) : base(c
        {
        }
    }
```

2. 领域服务的实现。具体的领域服务接口和实现代码如下所示：

```
// 领域服务接口
    public interface IDomainService
    {
        Order CreateOrder(User user, ShoppingCart shoppingCart);
    }

    // 领域服务类型
    // 牵涉到多个实体的操作可以把这些操作封装到领域服务里
    public class DomainService : IDomainService
    {
         private readonly IRepositoryContext _repositoryContext;
        private readonly IShoppingCartItemRepository _shoppingCartI
        private readonly IOrderRepository _orderRepository;

         /// <summary>
        /// 创建订单，涉及到的操作有2个：1\. 把购物车中的项中购物车移除； 2.
        /// 这两个操作必须同时完成或失败。
        /// </summary>
        /// <param name="user"></param>
        /// <param name="shoppingCart"></param>
        /// <returns></returns>
        public Order CreateOrder(User user, ShoppingCart shoppingCa
        {
            var order = new Order();
            var shoppingCartItems =
                _shoppingCartItemRepository.GetAll(
                    new ExpressionSpecification<ShoppingCartItem>(s
            if (shoppingCartItems == null || !shoppingCartItems.Any
                throw new InvalidOperationException("购物篮中没有任何

            order.OrderItems = new List<OrderItem>();
            foreach (var shoppingCartItem in shoppingCartItems)
            {
                var orderItem = shoppingCartItem.ConvertToOrderIter
                orderItem.Order = order;
                order.OrderItems.Add(orderItem);
                _shoppingCartItemRepository.Remove(shoppingCartIter
            }
            order.User = user;
            order.Status = OrderStatus.Paid;
            _orderRepository.Add(order);
            _repositoryContext.Commit();
            return order;
        }
    }
```

3. 订单服务的实现。具体订单服务实现代码如下所示：

```
public class OrderServiceImp : ApplicationService, IOrderService
    {
```

```
        #region Private Fileds
        private readonly IShoppingCartRepository _shoppingCartRepos
        private readonly IShoppingCartItemRepository _shoppingCartI
        private readonly IUserRepository _userRepository;
        private readonly IOrderRepository _orderRepository;
        private readonly IProductRepository _productRepository;
        private readonly IDomainService _domainService;
        private readonly IEventBus _eventBus;
        #endregion

        #region Ctor
        public OrderServiceImp(IRepositoryContext context,
            IUserRepository userRepository,
            IShoppingCartRepository shoppingCartRepository,
            IProductRepository productRepository,
            IShoppingCartItemRepository shoppingCartItemRepository,
            IDomainService domainService,
            IOrderRepository orderRepository,
            IEventBus eventBus) : base(context)
        {
            _userRepository = userRepository;
            _shoppingCartRepository = shoppingCartRepository;
            _productRepository = productRepository;
            _shoppingCartItemRepository = shoppingCartItemReposito
            _domainService = domainService;
            _orderRepository = orderRepository;
            _eventBus = eventBus;
        }

        #endregion

         public OrderDto Checkout(Guid customerId)
        {
            var user = _userRepository.GetByKey(customerId);
            var shoppingCart = _shoppingCartRepository.GetByExpress
            var order = _domainService.CreateOrder(user, shoppingCa

            return Mapper.Map<Order, OrderDto>(order);
        }

        public OrderDto GetOrder(Guid orderId)
        {
            var order = _orderRepository.GetBySpecification(new Exp
            return Mapper.Map<Order, OrderDto>(order);
        }

        // 获得指定用户的所有订单
        public IList<OrderDto> GetOrdersForUser(Guid userId)
        {
            var user = _userRepository.GetByKey(userId);
            var orders = _orderRepository.GetAll(new ExpressionSpec
            var orderDtos = new List<OrderDto>();
            orders
```

```
                .ToList()
                .ForEach(o=>orderDtos.Add(Mapper.Map<Order, OrderDt
            return orderDtos;
        }
```

4. HomeController控制器中Checkout操作的实现。具体实现代码如下所示：

```
public class HomeController : ControllerBase
{
        /// <summary>
        /// 结算操作
        /// </summary>
        /// <returns></returns>
        [Authorize]
        public ActionResult Checkout()
        {
            using (var proxy = new OrderServiceClient())
            {
                var model = proxy.Checkout(this.UserId);
                return View(model);
            }
        }

        [Authorize]
        public ActionResult Orders()
        {
            using (var proxy = new OrderServiceClient())
            {
                var model = proxy.GetOrdersForUser(this.UserId);
                return View(model);
            }
        }

        [Authorize]
        public ActionResult Order(string id)
        {
            using (var proxy = new OrderServiceClient())
            {
                var model = proxy.GetOrder(new Guid(id));
                return View(model);
            }
        }
}
```

这样我们就在网上书店中实现了订单功能了。具体的视图界面也就是上一专题中实现的购物车页面。下面具体看看订单的具体实现效果：

结算页面：



点击确认购买按钮，在弹出框中点击确认来完成订单的创建：

Online Store 首页 我的 关于 欢迎您，sssa 0个商品 退出

原版进口白菜价 海外订购 原知原味 300万种进口图书 低至5折

图书分类

- 所有图书
- MongoDB
- 领域驱动
- C#
- 操作系统
- No-SQL
- Java
- 算法
- Asp.net

生成订单成功！

订单编号：	48A46900-EE0D-E511-AD6F-206A8A06A4C1
收货地址：	20000, 张江镇 上海, 上海, 中国
联系人：	Learninghard
联系电话：	13327890340
电子邮件：	794170314@qq.com

您可以 单击此处 查看此订单的详细信息，或者 单击此处 查看所有订单。

请 单击此处 回到首页继续购物。

通过我的订单来查看所有订单页面：



四、总结

到此，网上书店案例的订单功能的实现就完成了，在接下来的专题将继续对该案例进行完善，在下一专题中将为该案例引入后台管理操作。商家或管理员可以进入后台管理来对用户订单进行确认发货，以及添加商品，分类等操作。具体实现请见下一专题。

本专题中所有实现源码下载：https://github.com/lizhi5753186/OnlineStore_Second/

[.NET领域驱动设计实战系列]专题七：DDD实践案例：引入事件驱动与中间件机制来实现后台管理功能

一、引言

在当前的电子商务平台中，用户下完订单之后，然后店家会在后台看到客户下的订单，然后店家可以对客户的订单进行发货操作。此时客户会在自己的订单状态看到店家已经发货。从上面的业务逻辑可以看出，当用户下完订单之后，店家或管理员可以对客户订单进行跟踪和操作。上一专题我们已经实现创建订单的功能，则接下来自然就是后台管理功能的实现了。所以在这一专题中将详细介绍如何在网上书店案例中实现后台管理功能。

二、后台管理中的权限管理的实现

后台管理中，首先需要实现的自然就是权限管理了，因为要进行商品管理等操作的话，则必须对不同的用户指定的不同角色，然后为不同角色指定不同的权限。这样才能确保普通用户不能进行一些后台操作。

然而角色和权限的赋予一般都是由系统管理员来操作。所以在最开始创建一个管理员用户，之后就可以以管理员的账号进行登录来进行后台操作的管理，包括添加角色，为用户分配角色、添加用户等操作。

这里就牵涉到一个权限管理的问题了。系统如何针对不同用户的全新进行管理呢？

其权限管理一个实现思路其实如下：

- 不同角色可以看到不同的链接，只有指定权限的用户才可以看到与其对应权限的操作。如只有管理员才可以添加用户和为用户赋予权限，而卖家只能对消费者订单的处理和对自己商店添加商品等操作。

从上面的描述可以发现，权限管理的实现主要包括两部分：

1. 为不同用户指定不同的链接显示。如管理员可以看到后台管理的所有链接：包括角色管理，商品管理，用户管理、订单管理，商品分类管理，而卖家只能看到订单管理，商品管理和商品类别管理等。其实现就是为这些链接的生成指定不同的权限，只有达到权限用户才进行生成该链接
2. 既然要为不同用户指定不同的权限，则首先要获得用户的权限，然后根据用户的权限来动态生成对应的链接。

有了上面的思路，下面就让我们一起为网上书店案例加入权限管理的功能：

首先，我在Layout.cshtml页面加入指定权限的链接，具体的代码如下所示：

```
<table width="996" border="0" cellspacing="0" cellpadding="0" aligr
        <tr>
            <td height="607" valign="top">
                <table width="996" border="0" cellspacing="0" cellp
                    <tr>
                        <td width="300" height="55" class="logo"><
                        <td width="480" class="menu">
                             <ul class="sf-menu">
                                <li>@Html.ActionLink("首页", "Index
                                @if (User.Identity.IsAuthenticated
                                {
                                <li>@Html.ActionLink("我的", "Manag
                                    <ul>
                                        <li>@Html.ActionLink("订单"
                                        <li>@Html.ActionLink("账户"
                                        <li>@Html.ActionLink("购物车
                                    </ul>
                                </li>
                                }
                               **@if (User.Identity.IsAuthenticatec
                                {** **<li>@Html.ActionLinkWithPermi
                                        <ul>
                                            <li>@Html.ActionLinkWit
                                            <li>@Html.ActionLinkWit
                                            <li>@Html.ActionLinkWit
                                            <li>@Html.ActionLinkWit
                                            <li>@Html.ActionLinkWit
                                        </ul>
                                    </li>
                                }** <li>@Html.ActionLink("关于", "A
                                    <ul>
                                        <li>@Html.ActionLink("Onlir
                                        <li>@Html.ActionLink("联系方
                                    </ul>
                                </li>
                            </ul>
                        </td>
                        <td width="216" class="menu">
                            @{Html.RenderAction("_LoginPartial", "L
                        </td>
                    </tr>
                </table>
                <table width="100%" border="0" cellspacing="0" cell
                    <tr>
                        <td width="100%" height="10px" />
                    </tr>
                </table>
                <table width="996" border="0" cellspacing="0" cellp
                    <tr>
                        <td>
                            <img src="/images/header.jpg" alt="" wi
                    </tr>
```

```
                </table>
                <table width="996" border="0" cellspacing="0" cellp
                    <tr align="left" valign="top">
                        <td width="202" height="334">
                            @{Html.RenderAction("CategoriesPartial'
                        </td>
                        <td width="20"> </td>
                        <td width="774">

                            <table width="774" border="0" cellspac:
                                <tr>
                                    @(MvcSiteMap.Instance.Navigator
                                </tr>
                                <tr>
                                    <td>
                                        @RenderBody()
                                    </td>
                                </tr>

                            </table>
                        </td>
                    </tr>
                </table>
                <table width="996" border="0" cellspacing="0" cellp
                    <tr>
                        <td>
                            <img src="/images/footer.jpg" alt="" w:
                    </tr>
                    <tr>
                        <td height="76">
                            <table width="996" border="0" cellspac:
                                <tr>
                                    <td width="329" height="78" al:
                                    <td width="14"> </td>
                                    <td width="653"><span class="st
                                        版权所有 &copy; 2014-2015, C
                                </tr>
                            </table>
                        </td>
                    </tr>
                </table>

            </td>
        </tr>
    </table>
```

上面红色加粗部分就是设置不同角色的不同权限。其中ActionLinkWithPermission是一个扩展方法，其具体实现就是获得登陆用户的角色，然后把用户的角色与当前的需要权限进行比较，如果相同，则通过HtmlHelper.GenerateLink方法来生成对应的链接。该方法的实现代码如下所示：

```
public static MvcHtmlString ActionLinkWithPermission(this HtmlHelp
        {
            if (helper == null ||
                helper.ViewContext == null ||
                helper.ViewContext.RequestContext == null ||
                helper.ViewContext.RequestContext.HttpContext == nu
                helper.ViewContext.RequestContext.HttpContext.User 
                helper.ViewContext.RequestContext.HttpContext.User.
                return MvcHtmlString.Empty;

            using (var proxy = new UserServiceClient())
            {
                var role = proxy.GetRoleByUserName(helper.ViewConte
                if (role == null)
                    return MvcHtmlString.Empty;
                var keyName = role.Name;
                var permissionKey = (PermissionKeys)Enum.Parse(type

                // 通过用户的角色和对应对应的权限进行与操作
                // 与结果等于用户角色时，表示用户角色与所需要的权限一样，则创
                return (permissionKey & required) == permissionKey 
                    MvcHtmlString.Create(HtmlHelper.GenerateLink(he
                    : MvcHtmlString.Empty;
            }
        }
```

通过上面的代码，我们就已经完成了权限管理的实现了。

三、后台管理中商品管理的实现

如果你是管理员的话，这样你就可以进入后台页面对商品、用户、订单等进行管理了。在上面我们已经完成了权限管理的实现。接下来，我们可以用一个管理员账号登陆之后，你可以看到管理员对应的权限。这里我直接在数据库中添加了一条管理员账号，其账号信息是admin，密码也是admin。下面我就这个账号后看到的界面如下图所示：



从上图可以看出，后台管理包括销售订单管理、商品类别管理、商品信息管理等。这些都是一些类似的实现，都是一些增、删、改功能的实现。这里就是商品信息管理为例来介绍下。点击商品信息管理后，将可以看到所有商品列表，在该页面可以进商品进行添加、修改和删除等操作。其实现主要是通过应用服务来调用仓储来实现商品的信息的持久化罢了，下面就具体介绍下商品添加功能的实现。因为商品的添加需要首先把上传的图片先添加到服务器上的Images文件夹下，然后通过控制器来调用ProductService的CreateProducts方法来把商品保存到数据库中。

首先是图片上传功能的实现，其实现代码如下所示：

```
 [HandleError]
    public class AdminController : ControllerBase
    {
        #region Common Utility Actions

        // 保存图片到服务器指定目录下
        [NonAction]
        private void SaveFile(HttpPostedFileBase postedFile, string
        {
            string phyPath = Request.MapPath("~" + filePath);
            if (!Directory.Exists(phyPath))
            {
                Directory.CreateDirectory(phyPath);
            }
            try
            {
                postedFile.SaveAs(phyPath + saveName);
            }
            catch (Exception e)
            {
                throw new ApplicationException(e.Message);

            }
        }

        // 图片上传功能的实现
        [HttpPost]
        public ActionResult Upload(HttpPostedFileBase fileData, str
        {
            var result = string.Empty;
            if (fileData != null)
            {
                string ext = Path.GetExtension(fileData.FileName);
                result = Guid.NewGuid()+ ext;
                SaveFile(fileData, Url.Content("~/Images/Products/"
            }
            return Content(result);
        }
}
```

图片上传成功之后，接下来点击保存按钮则把商品进行持久化到数据库中。其实现逻辑主要是调用商品仓储的实现类来完成商品的添加。主要的实现代码如下所示：

```
[HandleError]
  public class AdminController : ControllerBase
  {
      [HttpPost]
      [Authorize]
      public ActionResult AddProduct(ProductDto product)
      {
          using (var proxy = new ProductServiceClient())
          {
              if (string.IsNullOrEmpty(product.ImageUrl))
              {
                  var fileName = Guid.NewGuid() + ".png";
                  System.IO.File.Copy(Server.MapPath("~/Images/Pr
                  product.ImageUrl = fileName;
              }
              var addedProducts = proxy.CreateProducts(new List<P
              if (product.Category != null &&
                  product.Category.Id != Guid.Empty.ToString())
                  proxy.CategorizeProduct(new Guid(addedProducts[
              return RedirectToSuccess("添加商品信息成功！", "Produc
          }
      }
  }

  // 商品服务的实现
  public class ProductServiceImp : ApplicationService, IProductSe
  {
         public List<ProductDto> CreateProducts(List<ProductDto> 
      {
          return PerformCreateObjects<List<ProductDto>, ProductDt
      }
  }
```

到此，我们已经完成了商品添加功能的实现，下面让我们看看商品添加的具体效果如何。添加商品页面：

图书分类

- 所有图书
- MongoDB
- 领域驱动
- C#
- 操作系统
- No-SQL
- Java
- 算法
- Asp.net

添加商品信息

商品名称

C# 并发编程经典实例

商品说明

C# 并发编程经典实例

单价

44

分类名称

C#

商品图片

选择图片

是否为新商品？

True

保存更改 取消编辑

点击保存更改按钮后，则进行商品的添加，添加成功后界面效果：

图书分类

- 所有图书
- MongoDB
- 领域驱动
- C#
- 操作系统
- No-SQL
- Java
- 算法
- Asp.net

商品信息管理

请使用下面的工具按钮对商品信息进行增添、编辑或删除的管理操作。单击商品名称同样可以打开该分类的编辑页面。

	名称	描述	分类	单价		
	ASP.NET设计模式	《ASP.NET设计模式》涵盖了开发企业级ASP.NET应用程序的知名模式和最佳实践。	Asp.net	¥65.40		×
	C# 并发编程经典实例	C# 并发编程经典实例	C#	¥44.00		×
	CLR via C#（第4版）	《CLR via C#（第4版）》针对CLR和.NET Framework 4.5进行深入、全面的探讨，并结合实例介绍了如何利用它们进行设计、开发和调试。	C#	¥92.70		×
	HTML5权威指南	《图灵程序设计丛书：HTML5权威指南》是系统学习网页设计的权威参考图书。	Asp.net	¥91.00		×
	Learninghard C#学习笔记	《图灵原创：Learning hard C#学习笔记》全部是作者亲身学习经验的总结，超详尽的学习笔记，获博客园数万网友点赞推荐。	C#	¥41.00		×
		与传统的关系型数据库不同，				

四、后台管理中发货操作和确认收货的实现

当消费者创建订单之后，然后卖家或管理员可以通过订单管理页面来对订单进行发货处理操作。以通知购买者该商品已发货了。在当前的电子商务网站中，除了更新订单的状态外，还会发邮件或短信通知购买者。为了保证这两个操作同时完成，此

时需要将这两个放在同一个事务中进行提交。

这里为了使系统有更好地可扩展性，采用了基于消息队列和事件驱动的方式来完成发货操作。在看具体实现代码之前，我们先来分析下实现思路：

- 卖家或管理员在订单管理页面，点击发货按钮后，此时相当于订单的状态进行了更新，从已付款状态到已发货状态。这里当然你可以采用传统的方式来实现，即调用订单仓储来更新对应订单的状态。但是这样的实现方式，邮件发送操作可能会嵌套在应用服务层了。这样的设计显然不适合扩展。所以这里采用基于事件驱动和消息队列方式来改进这种方式。
  1. 首先，当商家点击发货操作，此时会产生一个发货事件；
  2. 接着由注册的领域事件处理程序进行对该领域事件处理，处理逻辑主要是更新订单的状态和更新时间；
  3. 然后再将该事件发布到EventBus，EventBus中保存了一个队列来存放事件，发布操作的实现就是往该队列插入一个待处理的事件；
  4. 最后在EventBus中的Commit方法中对队列中的事件进行出队列操作，通过事件聚合类来获得对应事件处理器来对出队列的事件进行处理。
- 事件聚合器通过Unity注入（应用）事件的处理器。在EventAggregator类中定义_eventHandlers来保存所有（应用）事件的处理器，在EventAggregator的构造函数中通过调用其Register方法把对应的事件处理器添加到_eventHandlers字典中。然后在EventBus中的Commit方法中通过找到EventAggregator中的Handle方法来触发事件处理器来处理对应事件，即发出邮件通知。这里事件聚合器起到映射的功能，映射应用事件到对应的事件处理器来处理。

通过上面的分析可以发现，发货操作和收货操作都涉及2类事件，一类是领域事件，另一类处于应用事件，领域事件的处理由领域事件处理器来处理，而应用事件的处理不能定义在领域层，所以我们这里新建了一个应用事件处理层，叫OnlineStore.Events.Handlers，已经新建了一个对EventBus支持的层，叫OnlineStore.Events。经过上面的分析，实现发货操作和收货操作是不是有点清晰了呢？如果不是的话也没关系，我们可以结合下面具体的实现代码再来理解下上面分析的思路。因为收货操作和发货操作的实现非常类似，这里只贴出发货操作实现的主要代码进行演示。

首先是AdminController中DispatchOrder操作的实现：

```
public ActionResult DispatchOrder(string id)
        {
            using (var proxy = new OrderServiceClient())
            {
                proxy.Dispatch(new Guid(id));
                return RedirectToSuccess(string.Format("订单 {0} 已
            }
        }
```

接下来便是OrderService中Dispatch方法的实现：

```
public void Dispatch(Guid orderId)
        {
            using (var transactionScope = new TransactionScope())
            {
                var order = _orderRepository.GetByKey(orderId);
                order.Dispatch();
                _orderRepository.Update(order);
                RepositorytContext.Commit();
                _eventBus.Commit();
                transactionScope.Complete();
            }
        }
```

下面是Order实体类中Dispatch方法的实现：

```
/// <summary>
        /// 处理发货。
        /// </summary>
        public void Dispatch()
        {
            // 处理领域事件
            DomainEvent.Handle<OrderDispatchedEvent>(new OrderDispa
        }
```

接下来便是领域事件中Handle方法的实现了，其实现逻辑就是获得所有已注册的领域事件处理器，然后分别事件处理器进行调用。具体的实现代码如下所示：

```
public static void Handle<TDomainEvent>(TDomainEvent domainEvent)
            where TDomainEvent : class, IDomainEvent
        {
            // 找到对应的事件处理器来对事件进行处理
            var handlers = ServiceLocator.Instance.ResolveAll<IDoma
            foreach (var handler in handlers)
            {
                if (handler.GetType().IsDefined(typeof(HandlesAsync
                    Task.Factory.StartNew(() => handler.Handle(doma
                else
                    handler.Handle(domainEvent);
            }
        }
```

对应OrderDispatchedEventHandler类中Handle方法的实现如下所示：

```
// 发货事件处理器
    public class OrderDispatchedEventHandler : IDomainEventHandler<
    {
        private readonly IEventBus _bus;

        public OrderDispatchedEventHandler(IEventBus bus)
        {
            _bus = bus;
        }

        public void Handle(OrderDispatchedEvent @event)
        {
            // 获得事件源对象
            var order = @event.Source as Order;
            // 更新事件源对象的属性
            if (order == null) return;

            order.DispatchedDate = @event.DispatchedDate;
            order.Status = OrderStatus.Dispatched;

            // 这里把领域事件认为是一种消息，推送到EventBus中进行进一步处理
            _bus.Publish<OrderDispatchedEvent>(@event);
        }
    }
```

从上面代码中可以发现，领域事件处理器中只是简单更新订单状态的状态为Dispatched和更新订单发货时间，之后就把该事件继续发布到EventBus中进一步进行处理。EventBus类的具体实现代码如下所示：

```
// 领域事件处理器只是对事件对象的状态进行更新
    // 后续的事件处理操作交给EventBus进行处理
    // 本案例中EventBus主要处理的任务就是发送邮件通知，
    // 在EventBus一般处理应用事件，而领域事件处理器一般处理领域事件
    public class EventBus : DisposableObject, IEventBus
    {
            public EventBus(IEventAggregator aggregator)
        {
            this._aggregator = aggregator;

            // 获得EventAggregator中的Handle方法
            _handleMethod = (from m in aggregator.GetType().GetMeth
                                 let parameters = m.GetParameters()
                                 let methodName = m.Name
                                 where methodName == "Handle" &&
                                 parameters != null &&
                                 parameters.Length == 1
                                 select m).First();
        }

             public void Publish<TMessage>(TMessage message)
            where TMessage : class, IEvent
        {
            _messageQueue.Value.Enqueue(message);
            _committed.Value = false;
        }

        // 触发应用事件处理器对事件进行处理
        public void Commit()
        {
            while (_messageQueue.Value.Count > 0)
            {
                var evnt = _messageQueue.Value.Dequeue();
                var evntType = evnt.GetType();
                var method = _handleMethod.MakeGenericMethod(evntTy
                // 调用应用事件处理器来对应用事件进行处理
                method.Invoke(_aggregator, new object[] { evnt });
            }
            _committed.Value = true;
        }
     }
```

其EventAggregator类的实现如下所示：

```
public class EventAggregator : IEventAggregator
    {
        private readonly object _sync = new object();
        private readonly Dictionary<Type, List<object>> _eventHandl
        private readonly MethodInfo _registerEventHandlerMethod;
```

```
        public EventAggregator()
        {

            // 通过反射获得EventAggregator的Register方法
            _registerEventHandlerMethod = (from p in this.GetType(
                                            let methodName = p.Name
                                            let parameters = p.GetPa
                                            where methodName == "Reg
                                            parameters != null &&
                                            parameters.Length == 1 &
                                            parameters[0].ParameterT
                                            select p).First();
        }

        public EventAggregator(object[] handlers)
            : this()
        {
            // 遍历注册的EventHandler来把配置文件中具体的EventHanler通过
            foreach (var obj in handlers)
            {
                var type = obj.GetType();
                var implementedInterfaces = type.GetInterfaces();
                foreach (var implementedInterface in implementedIn
                {
                    if (implementedInterface.IsGenericType &&
                        implementedInterface.GetGenericTypeDefinit
                    {
                        var eventType = implementedInterface.GetGe
                        var method = _registerEventHandlerMethod.M
                        // 调用Register方法将EventHandler添加进_event
                        method.Invoke(this, new object[] { obj });
                    }
                }
            }
        }

        public void Register<TEvent>(IEventHandler<TEvent> eventHa
            where TEvent : class, IEvent
        {
            lock (_sync)
            {
                var eventType = typeof(TEvent);
                if (_eventHandlers.ContainsKey(eventType))
                {
                    var handlers = _eventHandlers[eventType];
                    if (handlers != null)
                    {
                        handlers.Add(eventHandler);
                    }
                    else
                    {
                        handlers = new List<object> {eventHandler}
                    }
```

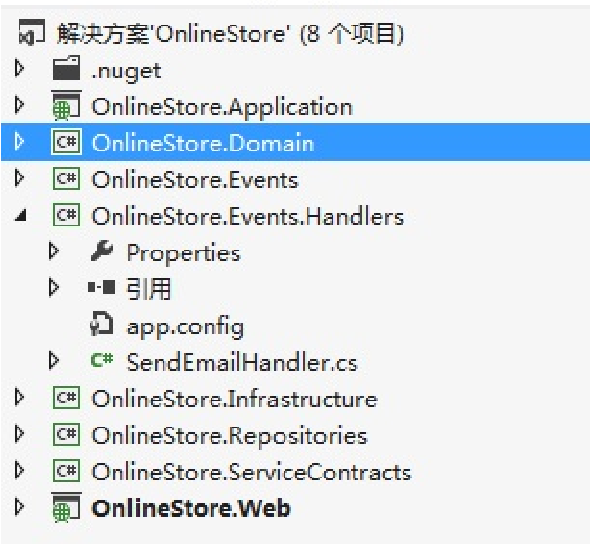
```
                }
                else
                    _eventHandlers.Add(eventType, new List<object>
            }
        }

        public void Register<TEvent>(IEnumerable<IEventHandler<TEve
            where TEvent : class, IEvent
        {
            foreach (var eventHandler in eventHandlers)
                Register<TEvent>(eventHandler);
        }

        // 调用具体的EventHanler的Handle方法来对事件进行处理
         public void Handle<TEvent>(TEvent evnt)
            where TEvent : class, IEvent
        {
            if (evnt == null)
                throw new ArgumentNullException("evnt");
            var eventType = evnt.GetType();
            if (_eventHandlers.ContainsKey(eventType) &&
                _eventHandlers[eventType] != null &&
                _eventHandlers[eventType].Count > 0)
            {
                var handlers = _eventHandlers[eventType];
                foreach (var handler in handlers)
                {
                    var eventHandler = handler as IEventHandler<TEv
                    if(eventHandler == null)
                        continue;

                    // 异步处理
                    if (eventHandler.GetType().IsDefined(typeof(Han
                    {
                        Task.Factory.StartNew((o) => eventHandler.H
                    }
                    else
                    {
                        eventHandler.Handle(evnt);
                    }
                }
            }
        }
```

至于确认收货操作的实现也是类似，大家可以自行参考Github源码进行实现。到此，我们商品发货和确认收货的功能就实现完成了。此时，我们解决方案已经调整为：



经过本专题后，我们网上书店案例的业务功能都完成的差不多了，后面添加的一些功能都是附加功能，例如分布式缓存的支持、分布式消息队列的支持以及面向切面编程的支持等功能。既然业务功能都完成的差不多了，下面让我们具体看看发货操作的实现效果吧。

首先是销售订单管理首页，在这里可以看到所有用户的订单状态。具体效果如下图示所示：

Online Store　首页　我的　管理　关于　欢迎您，admin　0个商品　退出

原版进口白菜价
海外订购
原知原味
300万种进口图书
低至5折

图书分类
- 所有图书
- MongoDB
- 领域驱动
- C#
- 操作系统
- No-SQL
- Java
- 算法
- Asp.net

销售订单管理

编号	用户名	条目数	总金额	创建日期	发货日期	收货日期	当前状态	发货
48A46900-EE0D-...	sssa	1	65.40 元	2015/6/8	N/A	N/A	已付款	发货
F76658B9-CC0C-...	sssa	1	65.40 元	2015/6/7	N/A	N/A	已付款	发货
ED046A8E-B506-...	sssa	1	41.00 元	2015/5/30	N/A	2015/6/2	已收货	
5DA938BC-B306-...	sssa	1	65.40 元	2015/5/30	N/A	2015/6/2	已收货	
6C2A3164-1406-...	sssa	3	225.90 元	2015/5/29	N/A	2015/6/2	已收货	

点击上图的发货按钮后便可以完成商品发货操作，此时创建该订单的用户邮箱中会收到一份发货邮件通知，具体实现效果截图如下所示：

其确认收货操作实现的效果与发货操作的效果差不多，这里就不一一截图了，大家可以自行到github上下载源码进行运行查看。

五、总结

到这里，该专题的介绍的内容就结束。本专题主要介绍后台管理中权限管理的实现、商品管理、类别管理、角色管理、用户角色管理和订单管理等功能。正如上面所说的，到此，本网上书店的DDD案例一些业务功能都实现的差不多了，接下来需要完善的功能主要是一些附加功能，这些功能主要是为了提高网站的可扩展性和可伸缩性。这些主要包括缓存的支持、分布式消息队列的支持以及AOP的支持。在下一个专题将介绍分布式缓存和分布式消息队列的支持，请大家继续关注。

本专题的所有源码下载：https://github.com/lizhi5753186/OnlineStore_Second/

[.NET领域驱动设计实战系列]专题八：DDD案例：网上书店分布式消息队列和分布式缓存的实现

一、引言

在上一专题中，商家发货和用户确认收货功能引入了消息队列来实现的，引入消息队列的好处可以保证消息的顺序处理，并且具有良好的可扩展性。但是上一专题消息队列是基于内存中队列对象来实现，这样实现有一个弊端，就是一旦服务重启或出现故障时，此时消息队列中的消息会丢失，并且也记录不了日志。所以就会出现，商家发货成功后，用户并没有收到邮件通知，并且也没有日志让我们发现是否发送了邮件通知。为了解决这个问题，就需要引入一种可恢复的消息队列。目前有很多开源的消息队列都支持可恢复的，例如TibcoEms.net等。然而，微软的MSMQ也是支持这种特性的。并且MSMQ还支持分布式部署，关于MSMQ更多内容可以参考：http://www.cnblogs.com/zhili/p/MSMQ.html

在本专题中将介绍为网上书店案例引入分布式消息队列和分布式缓存的实现。

二、分布式消息队列的实现

MSMQ的实现原理是：消息的发送者把自己想要发送的信息放入一个容器，然后把它保存到一个系统公用空间的消息队列中，本地或异地的消息接收程序再从该队列中取出发给它的消息进行处理。所以，即使服务器突然重启，消息也会存在于系统公用空间的消息队列中，待服务器重新启动后，可以继续接受消息进行处理，从而解决上一专题存在的问题。另外，上一专题的消息队列只能被用在当前服务器中，而MSMQ支持分布式部署，不同机器都可以对MSMQ进行接收消息来处理，此时MSMQ起到一个中间件的作用。

在为网上书店引入分布式消息队列之前，让我们先理一下实现思路：

- 上一专题中把发货事件和收货事件发布到EventBus中，而此时需要用MsmqEventBus来替代EventBus。而MsmqEventBus的实现就很简单了，完全可以参考EventBus来实现，只是此时消息并不是进入Queue对象中，而是通过MessageQueue.aspx%20)对象发送到系统的消息队列中。
- 而Commit方法即从系统的消息队列中出队来获得消息。再获得消息的处理器时，与上一专题的实现有点不同，因为把事件对象发送到消息队列时，需要先把事件对象先序列化为Message对象再放入消息队列中，而出队的也是消息对象，而不是上一专题中的发货事件对象。所以此时需要把出队的消息对象反序列化为对应的事件对象。

有了上面的实现思路，接下来让我们一起看看MsmqEventBus的具体实现代码吧。

```
public class MsmqEventBus : DisposableObject, IEventBus
    {
                public void Publish<TMessage>(TMessage message) where
```

```
        {
            // 将消息放入Message中Body属性进行序列化发送到消息队列中
            var msmqMessage = new Message(message) { Formatter = ne
            _messageQueue.Send(msmqMessage);
            _committed = false;
        }

        public void Publish<TMessage>(IEnumerable<TMessage> message
        {
            messages.ToList().ForEach(m =>
            {
                _messageQueue.Send(m);
                _committed = false;
            });
        }

        public void Commit()
        {
            if (this._useInternalTransaction)
            {
                using (var transaction = new MessageQueueTransactic
                {
                    try
                    {
                        transaction.Begin();
                        var message = _messageQueue.Receive();
                        if (message != null)
                        {
                            message.Formatter = new XmlMessageForma
                            var evntType = ConvertStringToType(mess
                            var method = _publishMethod.MakeGeneric
                            var evnt = Activator.CreateInstance(evn
                            method.Invoke(_aggregator, new object[]

                            transaction.Commit();
                        }
                    }
                    catch
                    {
                        transaction.Abort();
                        throw;
                    }
                }
            }
            else
            {
                // 从msmq消息队列中出队，此时获得的对象是消息对象
                var message = _messageQueue.Receive();
                if (message != null)
                {
                    // 指定反序列化的对象，由于我们之前把对应的事件类型保存
                    // 所以此时可以通过Label属性来获得目标序列化类型
                    message.Formatter = new XmlMessageFormatter(new
```

```
                    // 这样message.Body获得就是对应的事件对象，后面的处理
                    var evntType =message.Body.GetType();
                    var method = _publishMethod.MakeGenericMethod(e
                    method.Invoke(_aggregator, new object[] { messa
                }
            }

            _committed = true;
        }
    }
```

结合上面代码的注释和前面实现思路的介绍，相信理解MsmqEventBus应该没什么问题了。接下来，我们需要在配置文件中指定EventBus为MsmqEventBus类，另外需要在你本地专有队列中创建**"OnlineStoreQueue"**队列来接受消息。具体的配置文件修改为：

```
  <!--Event Bus-->
      <!--<register type="OnlineStore.Events.Bus.IEventBus, OnlineS
        <lifetime type="singleton" />
      </register>-->

      <!--注入MsmqEventBus-->
      <register type="OnlineStore.Events.Bus.IEventBus, OnlineStore
                mapTo="OnlineStore.Events.Bus.MsmqEventBus, OnlineS
        <lifetime type="singleton" />
        <constructor>
          <param name="path" value=".\Private$\OnlineStoreQueue" />
        </constructor>
      </register>
    </container>
```

到此，分布式消息队列的实现就完成了，具体分布式消息队列的实现效果和上一专题使用EventBus的效果是一样的，这里就不再贴图了，大家可以自行下载源码查看。

三、缓存的实现

在实际开发过程中，缓存的实现是必不可少的，对于已经查询过的数据可以直接从缓存中进行读取返回给调用者，利用缓存不但可以加快响应速度，还能减轻数据库服务器的压力。在大型电子商务网站中，缓存的实现更是必不可少的功能。然而缓存的实现也有两种，一种是分布式缓存，另一种本地缓存。在大型网站中，更多实现的是分布式缓存，对于一些少用户的企业系统，可能才会使用到本地缓存。所以在本专题中，将在网上书店案例中对这两种缓存分别进行实现。

3.1 本地缓存的实现

首先，我们来介绍本地缓存的实现。由于这里需要实现两种缓存，根据面向接口编程原则，我们自然首先需要定义一个缓存接口，然后这两种具体缓存都需要实现该接口。针对缓存接口，无非是缓存数据的添加，移除，更新等操作，所以缓存接口的定义如下所示：

```
// 缓存接口的定义
    public interface ICacheProvider
    {
        /// <summary>
        /// 向缓存中添加一个对象
        /// </summary>
        /// <param name="key">缓存的键值</param>
        /// <param name="valueKey">缓存值的键值</param>
        /// <param name="value">缓存的对象</param>
        void Add(string key, string valueKey, object value);
        void Update(string key, string valueKey, object value);
        object Get(string key, string valueKey);
        void Remove(string key);
        bool Exists(string key);
        bool Exists(string key, string valueKey);
    }
```

在介绍本地缓存的实现之前，让我们先来思考下本地缓存的实现思路——就是在本地缓存类中定义一个字典对象，添加缓存就是往该字典插入键值对而已，其中key就是缓存数据对应的键值，value就是真正的缓存数据，如果缓存在字典中存在的话，就直接根据键值查找出缓存数据进行返回。

然而网上书店的本地缓存是基于Enterprise Library Caching库来实现的，其实现思路和我之前介绍的思路也是一样的，只不过此时字典对象不需要我们在类中定义，此时直接用Enterprise Library Caching库中定义的就好。有了上面的分析，本地缓存的实现理解起来也就不那么难了，具体本地缓存的实现代码如下所示：

```
// 表示基于Microsoft Patterns & Practices - Enterprise Library Cachi
    // 该类简单理解为对Enterprise Library Caching中的CacheManager封装
    // 该缓存实现不支持分布式缓存，更多信息参考:
    // http://stackoverflow.com/questions/7799664/enterpriselibrary
    public class EntLibCacheProvider : ICacheProvider
    {
        // 获得CacheManager实例，该实例的注册通过cachingConfiguration进
        private readonly ICacheManager _cacheManager = CacheFactory

        #region ICahceProvider

        public void Add(string key, string valueKey, object value)
        {
            Dictionary<string, object> dict = null;
            if (_cacheManager.Contains(key))
```

```
            {
                dict = (Dictionary<string, object>) _cacheManager[k
                dict[valueKey] = value;
            }
            else
            {
                dict = new Dictionary<string, object> { { valueKey,
            }

            _cacheManager.Add(key, dict);
        }

        public void Update(string key, string valueKey, object valu
        {
            Add(key, valueKey, value);
        }

        public object Get(string key, string valueKey)
        {
            if (!_cacheManager.Contains(key)) return null;
            var dict = (Dictionary<string, object>)_cacheManager[ke
            if (dict != null && dict.ContainsKey(valueKey))
                return dict[valueKey];
            else
                return null;
        }

        // 从缓存中移除对象
        public void Remove(string key)
        {
            _cacheManager.Remove(key);
        }

        // 判断指定的键值的缓存是否存在
        public bool Exists(string key)
        {
            return _cacheManager.Contains(key);
        }

        // 判断指定的键值和缓存键值的缓存是否存在
        public bool Exists(string key, string valueKey)
        {
            return _cacheManager.Contains(key) &&
               ((Dictionary<string, object>)_cacheManager[key]).Con
        }
        #endregion
    }
```

到此，网上书店案例中本地缓存的实现就完成了。由于本地缓存不支持分布式部署，所有的缓存都存在于单独缓存服务器中，然而，针对一些大型网站来说，这样的实现并不适合，因为在大型网站中，需要通过多个缓存服务进行集群，需要使得

缓存均匀分布在集群中的缓存服务器中。此时就需要引入分布式缓存的实现。下面让我们具体看看分布式缓存如何在该案例中实现。

3.2 分布式缓存的实现

分布式缓存可以通过具体的算法把缓存均匀地分布在集群中缓存服务器中，从而用户请求的不同数据可以路由到对应的缓存服务器中进行添加、更新或获得。分布式缓存的实现有很多种方式，可以利用Memcached和Redis开源库来实现。然而，微软的Windows Azure也提供了分布式缓存的实现，本案例中分布式缓存就是基于Windows Azure的。在对分布式缓存实现之前，需要先下载对应的dll，然后再在项目中进行引用。需要下载的dll已经包含在项目根目录下的libs文件夹下，具体需要下载的程序集截图如下所示：

Microsoft.ApplicationServer.Caching.Client.dll	2014/4/19 8:51	应用程序扩展	170 KB
Microsoft.ApplicationServer.Caching.Core.dll	2014/4/19 8:51	应用程序扩展	818 KB
Microsoft.WindowsFabric.Common.dll	2014/4/19 8:51	应用程序扩展	186 KB
Microsoft.WindowsFabric.Data.Common.dll	2014/4/19 8:51	应用程序扩展	62 KB

然后在基础设施层引入这些程序集，之前就可以去实现基于Windows Azure的分布式缓存了。具体的实现代码如下所示：

```
// 分布式缓存，该类是对微软分布式缓存服务的封装
    // 在该案例中没用用到该缓存，但是提供在这里让大家明白微软的分布式缓存实现，
    // Redis参考：http://www.cnblogs.com/ceecy/p/3279407.html 和 htt
    // 关于微软分布式缓存更多介绍参考：http://www.cnblogs.com/shanyou/ar
    // 和http://www.cnblogs.com/mlj322/archive/2010/04/05/1704624.h
    public class AppfabricCacheProvider : ICacheProvider
    {
        private readonly DataCacheFactory _factory = new DataCacheF
        private readonly DataCache _cache;

        public AppfabricCacheProvider()
        {
            this._cache = _factory.GetDefaultCache();
        }

        #region ICacheProvider Members
        public void Add(string key, string valueKey, object value)
        {
            // DataCache中不包含Contain方法，所有用Get方法来判断对应的key
            var val = (Dictionary<string, object>)_cache.Get(key);
            if (val == null)
            {
                val = new Dictionary<string, object> {{ valueKey, v
                _cache.Add(key, val);
            }
            else
            {
                if (!val.ContainsKey(valueKey))
```

```
                    val.Add(valueKey, value);
                else
                    val[valueKey] = value;

                _cache.Put(key, val);
            }
        }

        public void Update(string key, string valueKey, object valu
        {
            Add(key, valueKey, value);
        }

        public object Get(string key, string valueKey)
        {
            return Exists(key, valueKey) ? ((Dictionary<string, obj
        }

        public void Remove(string key)
        {
            _cache.Remove(key);
        }

        public bool Exists(string key)
        {
            return _cache.Get(key) != null;
        }

        public bool Exists(string key, string valueKey)
        {
            var val = _cache.Get(key);
            if (val == null)
                return false;
            return ((Dictionary<string, object>)val).ContainsKey(va
        }

        #endregion
    }
```

通过上面的步骤，分布式缓存的实现就完成了。其实，分布式缓存和本地缓存不同之处就在于：分布式缓存支持对应的算法可以把缓存存放在不同的服务器上，而本地缓存只能存在于本地，而不能跨机器分布。所以对于大型网站，分布式缓存才是最好的选择，由于分布式缓存的实现和部署，无疑会增加开发和维护成本，所以对于一些小型系统（指定是单数据库服务器系统），可以考虑使用本地缓存。

在本案例中，由于本人没有Windows Azure环境，所以对于分布式缓存的实现也不能进行测试，所以本案例中使用的还是本地缓存。要使缓存生效，还需要对配置文件进行修改。具体配置文件修改为：

```
  <unity xmlns="http://schemas.microsoft.com/practices/2010/unity">
     <sectionExtension type="Microsoft.Practices.Unity.InterceptionE
     <container>
       <extension type="Interception" />

      **  <!--Cache Provider-->
       <register type="OnlineStore.Infrastructure.Caching.ICacheProv
 <!--.........-->
  </container>
 </unity>
```

其实，通过上面的配置之后，缓存还是不能生效的，因为我们一般把缓存放在获得数据方法之前进行调用，在用户对获得数据方法调用之前，首先从缓存中进行查找，如果存在，则直接返回缓存中的数据给调用者就可以了，如果不存在再调用获得数据方法从数据库中读取，读取成功后添加到缓存中再返回给调用者。既然要在方法调用前来查找缓存，从中你是否想到了什么呢？不错，就是面向切面编程，即AOP。所以要让缓存生效，在该案例中还需要支持AOP。至于AOP的支持，我将会在下一专题进行介绍。

四、总结

到这里，本专题的内容就结束了，正如前面所说的，在下一专题，我将在网上书店案例中引入对AOP的支持。

本专题所有源码下载地址：https://github.com/lizhi5753186/OnlineStore_Second/

[.NET领域驱动设计实战系列]专题九：DDD案例：网上书店AOP和站点地图的实现

一、引言

在前面一专题介绍到，要让缓存生效还需要实现对AOP（面向切面编程）的支持。所以本专题将介绍了网上书店案例中AOP的实现。关于AOP的概念，大家可以参考文章：http://www.cnblogs.com/jin-yuan/p/3811077.html。这里我简单介绍下AOP：AOP可以理解为对方法进行截获，这样就可以在方法调用前或调用后插入需要的逻辑。例如可以在方法调用前，加入缓存查找逻辑等。这里缓存查找逻辑就在方法调用前被执行。通过对AOP的支持，每个方法就可以分为3部分了，方法调用前逻辑->具体需要调用的方法->方法调用后的逻辑。也就是在方法调用的时候"切了一刀"。

二、网上书店AOP的实现

你可以从零开始去实现AOP，但是目前已经存在很多AOP框架了，所以在本案例中将直接通过Unity的AOP框架(Unity.Interception)来实现网上书店对AOP的支持。通常AOP的实现放在基础设施层进行实现，因为可能其他所有层都需要加入对AOP的支持。本案例中将对两个方面的AOP进行实现，一个是方法调用前缓存的记录或查找，另一个是方法调用后异常信息的记录。在实现具体代码之前，我们需要在基础设施层通过Nuget来引入Unity.Interception包。添加成功之后，我们需要定义两个类分别去实现AOP框架中**IInterceptionBehavior**接口。由于本案例中需要对缓存和异常日志功能进行AOP实现，自然就需要定义CachingBehavior和ExceptionLoggingBehavior两个类去实现**IInterceptionBehavior**接口。首先让我们看看CachingBehavior类的实现，具体实现代码如下所示：

```
// 缓存AOP的实现
    public class CachingBehavior : IInterceptionBehavior
    {
        private readonly ICacheProvider _cacheProvider;

        public CachingBehavior()
        {
            _cacheProvider = ServiceLocator.Instance.GetService<ICa
        }

        // 生成缓存值的键值
        private string GetValueKey(CacheAttribute cachingAttribute,
        {
            switch (cachingAttribute.Method)
            {
                // 如果是Remove，则不存在特定值键名，所有的以该方法名称相关
                case CachingMethod.Remove:
```

```
                    return null;
                // 如果是Get或者Update，则需要产生一个针对特定参数值的键名
                case CachingMethod.Get:
                case CachingMethod.Update:
                    if (input.Arguments != null &&
                        input.Arguments.Count > 0)
                    {
                        var sb = new StringBuilder();
                        for (var i = 0; i < input.Arguments.Count;
                        {
                            sb.Append(input.Arguments[i]);
                            if (i != input.Arguments.Count - 1)
                                sb.Append("_");
                        }

                        return sb.ToString();
                    }
                    else
                        return "NULL";
                default:
                    throw new InvalidOperationException("无效的缓存方
            }
        }

        #region IInterceptionBehavior Members
        public IEnumerable<Type> GetRequiredInterfaces()
        {
            return Type.EmptyTypes;
        }

        public IMethodReturn Invoke(IMethodInvocation input, GetNex
        {
            // 获得被拦截的方法
            var method = input.MethodBase;
            var key = method.Name; // 获得拦截的方法名
            // 如果拦截的方法定义了Cache属性，说明需要对该方法的结果需要进行
            if (!method.IsDefined(typeof (CacheAttribute), false))
                return getNext().Invoke(input, getNext);

            var cachingAttribute = (CacheAttribute)method.GetCustom
            var valueKey = GetValueKey(cachingAttribute, input);
            switch (cachingAttribute.Method)
            {
                case CachingMethod.Get:
                    try
                    {
                        // 如果缓存中存在该键值的缓存，则直接返回缓存中的结
                        if (_cacheProvider.Exists(key, valueKey))
                        {
                            var value = _cacheProvider.Get(key, val
                            var arguments = new object[input.Argume
                            input.Arguments.CopyTo(arguments, 0);
                            return new VirtualMethodReturn(input, v
```

```
                        }
                        else // 否则先调用方法，再把返回结果进行缓存
                        {
                            var methodReturn = getNext().Invoke(inp
                            _cacheProvider.Add(key, valueKey, metho
                            return methodReturn;
                        }
                    }
                    catch (Exception ex)
                    {
                        return new VirtualMethodReturn(input, ex);
                    }
                case CachingMethod.Update:
                    try
                    {
                        var methodReturn = getNext().Invoke(input,
                        if (_cacheProvider.Exists(key))
                        {
                            if (cachingAttribute.IsForce)
                            {
                                _cacheProvider.Remove(key);
                                _cacheProvider.Add(key, valueKey, r
                            }
                            else
                                _cacheProvider.Update(key, valueKey
                        }
                        else
                            _cacheProvider.Add(key, valueKey, metho

                        return methodReturn;
                    }
                    catch (Exception ex)
                    {
                        return new VirtualMethodReturn(input, ex);
                    }
                case CachingMethod.Remove:
                    try
                    {
                        var removeKeys = cachingAttribute.Correspor
                        foreach (var removeKey in removeKeys)
                        {
                            if (_cacheProvider.Exists(removeKey))
                                _cacheProvider.Remove(removeKey);
                        }
                        **// 执行具体截获的方法** var methodReturn = g
                        return methodReturn;
                    }
                    catch (Exception ex)
                    {
                        return new VirtualMethodReturn(input, ex);
                    }
                default: break;
            }
```

```
            return getNext().Invoke(input, getNext);
        }

        public bool WillExecute
        {
            get { return true; }
        }
        #endregion
    }
```

从上面代码可以看出，通过Unity.Interception框架来实现AOP变得非常简单了，我们只需要实现IInterceptionBehavior接口中的Invoke方法和WillExecute属性即可。并且从上面代码可以看出，AOP的支持最核心代码实现在于Invoke方法的实现。既然我们需要在方法调用前查找缓存，如果缓存不存在再调用方法从数据库中进行查找，如果存在则直接从缓存中进行读取数据即可。自然需要在getNext().Invoke(input, getNext)代码执行前进缓存进行查找，然而上面CachingBehavior类正式这样实现的。

介绍完缓存功能AOP的实现之后，下面具体看看异常日志的AOP实现。具体实现代码如下所示：

```
// 用于异常日志记录的拦截行为
    public class ExceptionLoggingBehavior :IInterceptionBehavior
    {
        /// <summary>
        /// 需要拦截的对象类型的接口
        /// </summary>
        /// <returns></returns>
        public IEnumerable<Type> GetRequiredInterfaces()
        {
            return Type.EmptyTypes;
        }

        /// <summary>
        /// 通过该方法来拦截调用并执行所需要的拦截行为
        /// </summary>
        /// <param name="input">调用拦截目标时的输入信息</param>
        /// <param name="getNext">通过行为链来获取下一个拦截行为的委托</p
        /// <returns>从拦截目标获得的返回信息</returns>
        public IMethodReturn Invoke(IMethodInvocation input, GetNex
        {
            // 执行目标方法
            var methodReturn = getNext().Invoke(input, getNext);
            // 方法执行后的处理
            if (methodReturn.Exception != null)
            {
                Utils.Log(methodReturn.Exception);
            }

            return methodReturn;
        }

        // 表示当拦截行为被调用时，是否需要执行某些操作
        public bool WillExecute
        {
            get { return true; }
        }
    }
```

异常日志功能的AOP实现与缓存功能的AOP实现类似，只是一个需要在方法执行前注入，而一个是在方法执行后进行注入罢了，其实现原理都是在截获的方法前后进行。方法截获功能AOP框架已经帮我们实现了。

到此，我们网上书店AOP的实现就已经完成了，但要正式生效还需要通过配置文件把AOP的实现注入到需要截获的方法当中去，这样执行这些方法才会执行注入的行为。对应的配置文件如下标红部分所示：

```
 <!--Unity的配置信息-->
  <unity xmlns="http://schemas.microsoft.com/practices/2010/unity">
    <sectionExtension type="Microsoft.Practices.Unity.InterceptionE
```

```
    <container>
      <extension type="Interception" />

       <!--Cache Provider-->
      <register type="OnlineStore.Infrastructure.Caching.ICacheProv

      <!--仓储接口的注册-->
      <register type="OnlineStore.Domain.Repositories.IRepositoryCo
          <lifetime type="singleton" />
      </register>
      <register type="OnlineStore.Domain.Repositories.IProductRepos
      <register type="OnlineStore.Domain.Repositories.ICategoryRepo
      <register type="OnlineStore.Domain.Repositories.IProductCateg
      <register type="OnlineStore.Domain.Repositories.IUserReposito
      <register type="OnlineStore.Domain.Repositories.IShoppingCart
      <register type="OnlineStore.Domain.Repositories.IShoppingCart
      <register type="OnlineStore.Domain.Repositories.IOrderReposit
      <register type="OnlineStore.Domain.Repositories.IUserReposito
      <register type="OnlineStore.Domain.Repositories.IUserRoleRepo
      <register type="OnlineStore.Domain.Repositories.IRoleReposito

       <!--Domain Services-->
      <register type="OnlineStore.Domain.Services.IDomainService, O
     ** <!--应用服务的注册-->
      <register type="OnlineStore.ServiceContracts.IProductService,
        <!--注入AOP功能的实现-->
        <interceptor type="InterfaceInterceptor" />
        <interceptionBehavior type="OnlineStore.Infrastructure.Inte
        <interceptionBehavior type="OnlineStore.Infrastructure.Inte
      </register>

      <register type="OnlineStore.ServiceContracts.IOrderService, O
        <!--注入AOP功能的实现-->
        <interceptor type="InterfaceInterceptor" />
        <interceptionBehavior type="OnlineStore.Infrastructure.Inte
        <interceptionBehavior type="OnlineStore.Infrastructure.Inte
      </register>
      <register type="OnlineStore.ServiceContracts.IUserService, On
        <!--注入AOP功能的实现-->
        <interceptor type="InterfaceInterceptor" />
        <interceptionBehavior type="OnlineStore.Infrastructure.Inte
        <interceptionBehavior type="OnlineStore.Infrastructure.Inte
      </register>**

      <!--Domain Event Handlers-->
      <register type="OnlineStore.Domain.Events.IDomainEventHandler
      <register type="OnlineStore.Domain.Events.IDomainEventHandler

      <!--Event Handlers-->
      <register name="orderSendEmailHandler" type="OnlineStore.Even

      <!--Event Aggregator-->
      <register type="OnlineStore.Events.IEventAggregator, OnlineSt
```

```
        <constructor>
          <param name="handlers">
            <array>
              <dependency name="orderSendEmailHandler" type="Online
            </array>
          </param>
        </constructor>
      </register>

      <!--Event Bus-->
      <!--<register type="OnlineStore.Events.Bus.IEventBus, OnlineS
        <lifetime type="singleton" />
      </register>-->

      <!--注入MsmqEventBus-->
      <register type="OnlineStore.Events.Bus.IEventBus, OnlineStore
                mapTo="OnlineStore.Events.Bus.MsmqEventBus, OnlineS
        <lifetime type="singleton" />
        <constructor>
          <param name="path" value=".\Private$\OnlineStoreQueue" />
        </constructor>
      </register>
    </container>
  </unity>
  <!--END: Unity-->
```

到此，网上书店案例中AOP的实现就完成了。通过上面的配置可以看出，客户端在调用应用服务方法前后会调用我们注入的行为，即缓存行为和异常日志行为。通过对AOP功能的支持，就不需要为每个需要进行缓存或需要异常日志行为的方法来重复写这些相同的逻辑了。从而避免了重复代码的重复实现，提高了代码的重用性和降低了模块之间的依赖性。

三、网上书店案例中站点地图的实现

在大部分网站中都实现了站点地图的功能，在Asp.net中，我们可以通过SiteMap模块来实现站点地图的功能，在Asp.net MVC也可以通过MvcSiteMapProvider第三方开源框架来实现站点地图。所以针对网上书店案例，站点地图的支持也是必不可少的。下面让我们具体看看站点地图在本案例中是如何去实现的呢？

在看实现代码之前，让我们先来理清下实现思路。

本案例中站点地图的实现，并没有借助MvcSiteMapProvider第三方框架来实现。其实现原理首先获得用户的路由请求，然后根据用户请求根据站点地图的配置获得对应的配置节点，接着根据站点地图的节点信息生成类似">首页>"这样带标签的字符串；如果获得的节点是配置文件中某个父节点的子节点，此时会通过递归的方式找到其父节点，然后递归地生成对应带标签的字符串，从而完成站点地图的功能。分析完实现思路之后，下面让我们再对照下具体的实现代码来加深理解。具体的实现代码如下所示：

```
public class MvcSiteMap
    {
        private static readonly MvcSiteMap _instance = new MvcSiteM

        private static readonly XDocument Doc = XDocument.Load(Http

        private UrlHelper _url = null;

        private string _currentUrl;

        public static MvcSiteMap Instance
        {
            get { return _instance;}
        }

        private MvcSiteMap()
        {
        }

        public MvcHtmlString Navigator()
        {
            // 获得当前请求的路由信息
            _url = new UrlHelper(HttpContext.Current.Request.Reques
            var routeUrl = _url.RouteUrl(HttpContext.Current.Reques
            if (routeUrl != null)
                _currentUrl = routeUrl.ToLower();

            // 从配置的站点Xml文件中找到当前请求的Url相同的节点
            var c = FindNode(Doc.Root);
            var temp = GetPath(c);

            return MvcHtmlString.Create(BuildPathString(temp));
        }

        // 从SitMap配置文件中找到当前请求匹配的节点
        private XElement FindNode(XElement node)
        {
            // 如果xml节点对应的url是否与当前请求的节点相同，如果相同则直接
            // 如果不同开始递归子节点
            return IsUrlEqual(node) == true ? node : RecursiveNode(
        }

        // 判断xml节点对应的url是否与当前请求的url一样
        private bool IsUrlEqual(XElement c)
        {
            var a = GetNodeUrl(c).ToLower();
            return a == _currentUrl;
        }

        // 递归Xml节点
        private XElement RecursiveNode(XElement node)
        {
```

```
            foreach (var c in node.Elements())
            {
                if (IsUrlEqual(c) == true)
                {
                    return c;
                }
                else
                {
                    var x = RecursiveNode(c);
                    if (x != null)
                    {
                        return x;
                    }
                }
            }

            return null;
        }

        // 获得xml节点对应的请求url
        private string GetNodeUrl(XElement c)
        {
            return _url.Action(c.Attribute("action").Value, c.Attri
                new {area = c.Attribute("area").Value});
        }

        // 根据对应请求url对应的Xml节点获得其在Xml中的路径，即获得其父节点有
        // SiteMap.xml 中节点的父子节点一定要配置对
        private Stack<XElement> GetPath(XElement c)
        {
            var temp = new Stack<XElement>();
            while (c != null)
            {
                temp.Push(c);
                c = c.Parent;
            }
            return temp;
        }

        // 根据节点的路径来拼接带标签的字符串
        private string BuildPathString(Stack<XElement> m)
        {
            var sb = new StringBuilder();
            var tc = new TagBuilder("span");
            tc.SetInnerText(">");
            var sp = tc.ToString();
            var count = m.Count;
            for (var x = 1; x <= count; x++)
            {
                var c = m.Pop();
                TagBuilder tb;
                if (x == count)
                {
```

```
                    tb = new TagBuilder("span");
                }
                else
                {
                    tb = new TagBuilder("a");
                    tb.MergeAttribute("href", GetNodeUrl(c));
                }

                tb.SetInnerText(c.Attribute("title").Value);
                sb.Append(tb);
                sb.Append(sp);
            }

            return sb.ToString();
        }
    }
```

对应的站点地图配置信息如下所示：

```
<?xml version="1.0" encoding="utf-8" ?>
<node title="首页" area="" action="Index" controller="Home">
  <node title="我的" area="UserCenter" action="Manage" controller="
    <node title="订单" area="" action="Orders" controller="Home" />
    <node title="账户" area="" action="Manage" controller="Account"
    <node title="购物车" area="" action="ShoppingCart" controller="H
  </node>
  <node title="关于" area="AboutCenter" action="About" controller="
    <node title="Online Store 项目" area="" action="About" controll
    <node title="联系方式" area="" action="Contact" controller="Home
  </node>
</node>
```

实现完成之后，下面让我们具体看看本案例中站点地图的实现效果看看，具体运行效果如下图所示：



四、小结

到这里，本专题的内容就结束了。本专题主要借助Unity.Interception框架在网上书店中引入了AOP功能，并且最后简单介绍了站点地图的实现。在下一专题将对CQRS模式做一个全面的介绍。

本案例所有源码：https://github.com/lizhi5753186/OnlineStore_Second/

[.NET领域驱动设计实战系列]专题十：DDD扩展内容：全面剖析CQRS模式实现

一、引言

前面介绍的所有专题都是基于经典的领域驱动实现的，然而，领域驱动除了经典的实现外，还可以基于CQRS模式来进行实现。本专题将全面剖析如何基于CQRS模式（Command Query Responsibility Segregation,命令查询职责分离）来实现领域驱动设计。

二、CQRS是什么？

在介绍具体的实现之前，对于之前不了解CQRS的朋友来说，首先第一个问题应该是：什么是CQRS啊？你倒是详细介绍完CQRS后再介绍具体实现啊？既然大家会有这样的问题，所以本专题首先全面介绍下什么是CQRS。

2.1 CQRS发展历程

在介绍CQRS之前，我觉得有必要先了解一下CQS（即Command Query Separation,命令查询分离）模式。我们可以理解CQRS是在DDD的实践中基于CQS理论而出现的一种体系结构模式。CQS模式最早由软件大师Bertrand Meyer（Eiffel语言之父，面向对象开-闭原则OCP提出者）提出，他认为，对象的行为仅有两种：命令和查询，不存在第三种情况。根据CQS的思想，任何方法都可以拆分为命令和查询两部分。例如下面的方法：

在上面的方法中，执行了一个命令，即对变量_number加上一个因子factor，同时又执行了一个查询，即查询返回_number的值。根据CQS的思想，该方法可以拆成Command和Query两个方法：

```
private int _number = 0;
private void AddCommand(int factor)
{
    _number += factor;
}

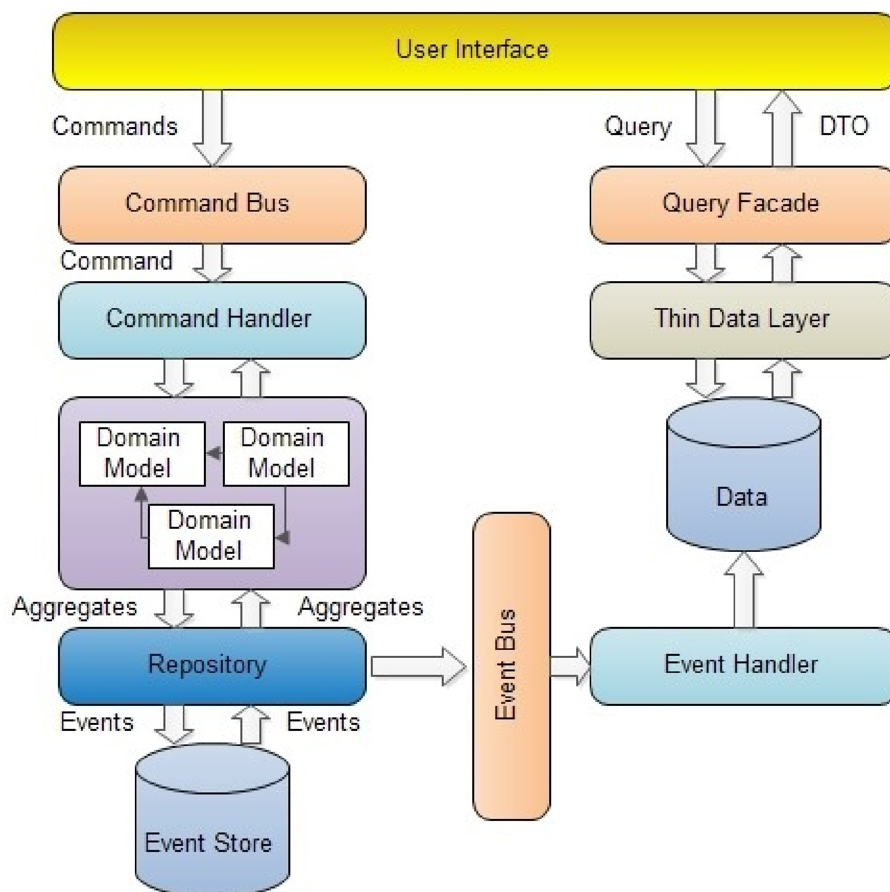
private int QueryValue()
{
    return _number;
}
```

命令和查询分离使得我们可以更好地把握对象的细节，更好地理解哪些操作会改变系统的状态。从而使的系统具有更好的扩展性，并获得更好的性能。

CQRS根据CQS思想，并结合领域驱动设计思想，由Grey Young在CQRS, Task Based UIs, Event Sourcing agh! 这篇文章中提出。**CQRS**将之前只需要定义一个对象拆分成两个对象，分离的原则按照对象中方法是执行命令还是执行查询来进行拆分的。

2.2 CQRS结构

由前面的介绍可知，采用CQRS模式实现的系统结构可以分为两个部分：命令部分和查询部分。其系统结构如下图所示：



从上面系统结构图可以发现，采用CQRS实现的领域驱动设计与经典DDD有很大的不同。采用CQRS实现的DDD结构大体分为两部分，查询部分和命令部分，并且维护着两个数据库实例，一个专门用来进行查询，另一个用来响应命令操作。然后通过EventHandler操作将命令改变的状态同步到用来查询的数据库实例中。从这个描述中，我们可能会联想到数据库级别主从读写分离。然而数据读写分离是在数据库层面来实现读写分离的机制，而CQRS是在业务逻辑层面来实现读写分离机制。两者是站在两个不同的层面对读写分离进行实现的。

三、为什么需要引入CQRS模式

前面我们已经详细介绍了CQRS模式，相信经过前面的介绍，大家对CQRS模式一定有一些了解了，但为什么要引入CQRS模式呢？

在传统的实现中，对DB执行增、删、改、查所有操作都会放在对应的仓储中，并且这些操作都公用一份领域实体对象。对于一些简单的系统，使用传统的设计方式并没有什么不妥，但在一些大型复杂的系统中，传统的实现方式也会存在一些问题：

- 使用同一个领域实体来进行数据读写可能会遇到资源竞争的情况。所以经常要处理锁的问题，在写入数据的时候，需要加锁，读取数据的时候需要判断是否允许脏读。这样使得系统的逻辑性和复杂性增加，并会影响系统的吞吐量。
- 在大数据量同时进行读写的情况下，可能出现性能的瓶颈。
- 使用同一个领域实体来进行数据库读写可能会太粗糙。在大多是情况下，比如编辑操作，可能只需要更新个别字段，这时却需要将整个对象都穿进去。还有在查询的时候，表现层可能只需要个别字段，但需要查询和返回整个领域实体，再把领域实体对象转换从对应的DTO对象。
- 读写操作都耦合在一起，不利于对问题的跟踪和分析，如果读写操作分离的话，如果是由于状态改变的问题就只需要去分析写操作相关的逻辑就可以了，如果是关于数据的不正确，则只需要关心查询操作的相关逻辑即可。

针对上面的这些问题，采用CQRS模式的系统都可以解决。由于CQRS模式中将查询和命令进行分析，所以使得两者分工明确，各自负责不同的部分，并且在业务上将命令和查询分离能够提高系统的性能和可扩展性。既然CQRS这么好，那是不是所有系统都应该基于CQRS模式去实现呢？显然不是的，CQRS也有其使用场景：

1. 系统的业务逻辑比较复杂的情况下。因为本来业务逻辑就比较复杂了，如果再把命令操作和查询操作绑定同一个业务实体的话，这样会导致后期的需求变更难于进行扩展下去。
2. 需要对系统中查询性能和写入性能分开进行优化的情况下，尤其读/写比例非常高的情况下。例如，在很多系统中读操作的请求数远大于写操作，此时，就可以考虑将写操作抽离出来进行单独扩展。
3. 系统在将来随着时间不断变化的情况下。

然而，CQRS也有其不适用的场景：

- 业务逻辑比较简单的情况下，此时采用CQRS反而会把系统搞的复杂。
- 系统用户访问量都比较小的情况下，并且需求以后不怎么会变更的情况下。针对这样的系统，完全可以用传统的实现方式快速将系统实现出来，没必要引入CQRS来增加系统的复杂度。

四、事件溯源

在**CQRS**中，查询方面，直接通过方法查询数据库，然后通过**DTO**将数据返回，这个方面的操作相对比较简单。而命令方面，是通过发送具体**Command**，接着由**CommandBus**来分发到具体的**CommandHandle**来进行处理，**CommandHandle**在进行处理时，并没有直接将对象的状态保存到外部持久化结构中，而仅仅是从领域对象中获得产生的一系列领域事件，并将这些事件保存到**Event Store**中，同时将事件发布到事件总线**Event Bus**进行下一步处理；接着**Event Bus**同样进行协调，将具体的事件交给具体的**Event Handle**进行处理，最后**Event Handler**再把对象的状态保存到对应**Query**数据库中。

上面过程正是CQRS系统中的调用顺序。从中可以发现，采用CQRS实现的系统存在两个数据库实例，一个是Event Store，该数据库实例用来保存领域对象中发生的一系列的领域事件，简单来说就是保存领域事件的数据库。另一个是Query Database，该数据库就是存储具体的领域对象数据的，查询操作可以直接对该数据库进行查询。由于，我们在Event Store中记录领域对象发生的所有事件，这样我们就可以通过查询该数据库实例来获得领域对象之前的所有状态了。所谓Event Sourcing，就是指的的是：通过事件追溯对象的起源，它允许通过记录下来的事件，将领域模型恢复到之前的任意一个时间点。

通过Event来记录领域对象所发生的所有状态，这样利用系统的跟踪并能够方便地回滚到某一历史状态。经过上面的描述，感觉事件溯源一般用于系统的维护。例如，我们可以设计一个同步服务，该服务程序从Event Store数据库查询出领域对象的历史数据，从而打印生成一个历史报表，如历史价格报表等。但正是的CQRS系统中如何使用Event Sourcing的呢？

在前面介绍CQRS系统的调用顺序中，我们讲到，由Event Handler将对象的状态保存到对应的Query数据库中，这里有一个问题，对象的状态怎么获得呢？对象状态的获得正是由Event sourcing机制来获得，因为用户发送的仅仅是Command，Command中并不包含对象的状态数据，所以此时需要通过Event Sourcing机制来查询Event Store来还原对象的状态，还原根据就是对应的Id，该Id是通过命令传入的。**Event Sourcing**的调用需要放在**CommandHandle**中，因为**CommandHandle**需要先获得领域对象，这样才能把领域对象与命令对象来进行对比，从而获得领域对象中产生的一系列领域事件。

五、快照

然而，当随着时间的推移，领域事件变得越来越多时，通过Event Sourcing机制来还原对象状态的过程会非常耗时，因为每一次都需要从最早发生的事件开始。那有没有好的一个方式来解决这个问题呢？答案是肯定的，即在Event Sourcing中引入快照（Snapshots）实现。实现原理就是——没产生N个领域事件，则对对象做一次快照。这样，领域对象溯源的时候，可以先从快照中获得最近一次的快照，然后再逐个应用快照之后所有产生的领域事件，而不需要每次溯源都从最开始的事件开始对对象重建，这样就大大加快了对象重建的过程。

六、CQRS模式实现和剖析

前面介绍了那么多CQRS的内容，下面就具体通过一个例子来演示下CQRS系统的实现。

命令部分的实现

```
// 应用程序初始化操作，将依赖的对象通过依赖注入框架StructureMap进行注入
public sealed class ServiceLocator
{
    private static readonly ICommandBus _commandBus;
    private static readonly IStorage _queryStorage;
    private static readonly bool IsInitialized;
```

```
        private static readonly object LockThis = new object();

        static ServiceLocator()
        {
            if (!IsInitialized)
            {
                lock (LockThis)
                {
                    // 依赖注入
                    ContainerBootstrapper.BootstrapStructureMap();

                    _commandBus = ContainerBootstrapper.Container.G
                    _queryStorage = ContainerBootstrapper.Containe
                    IsInitialized = true;
                }
            }
        }

        public static ICommandBus CommandBus
        {
            get { return _commandBus; }
        }

        public static IStorage QueryStorage
        {
            get { return _queryStorage; }
        }
    }

    class ContainerBootstrapper
    {
        private static Container _container;
        public static void BootstrapStructureMap()
        {
            _container = new Container(x =>
            {
                x.For(typeof (IDomainRepository<>)).Singleton().Use
                x.For<IEventStorage>().Singleton().Use<InMemoryEven
                x.For<IEventBus>().Use<EventBus>();
                x.For<ICommandBus>().Use<CommandBus>();
                x.For<IStorage>().Use<InMemoryStorage>();
                x.For<IEventHandlerFactory>().Use<StructureMapEvent
                x.For<ICommandHandlerFactory>().Use<StructureMapCom
            });
        }

        public static Container Container
        {
            get { return _container;}
        }
    }

public class HomeController : Controller
```

```
    {
         [HttpPost]
        public ActionResult Add(DiaryItemDto item)
        {
            // 发布CreateItemCommand到CommandBus中
            ServiceLocator.CommandBus.Send(new CreateItemCommand(Gu

            return RedirectToAction("Index");
        }
    }

 // CommandBus 的实现
    public class CommandBus : ICommandBus
    {
        private readonly ICommandHandlerFactory _commandHandlerFact

        public CommandBus(ICommandHandlerFactory commandHandlerFact
        {
            _commandHandlerFactory = commandHandlerFactory;
        }

        public void Send<T>(T command) where T : Command
        {
            // 获得对应的CommandHandle来对命令进行处理
            var handlers = _commandHandlerFactory.GetHandlers<T>();

            foreach (var handler in handlers)
            {
                // 处理命令
                handler.Execute(command);
            }
        }
    }

// 对CreateItemCommand处理类
    public class CreateItemCommandHandler : ICommandHandler<CreateI
    {
        private readonly IDomainRepository<DiaryItem> _domainReposi

        public CreateItemCommandHandler(IDomainRepository<DiaryIten
        {
            _domainRepository = domainRepository;
        }

        // 具体处理逻辑
        public void Execute(CreateItemCommand command)
        {
            if (command == null)
            {
                throw new ArgumentNullException("command");
            }
            if (_domainRepository == null)
            {
```

```
                throw new InvalidOperationException("domainReposito
            }

            var aggregate = new DiaryItem(command.ID, command.Title
            {
                Version = -1
            };

            // 将对应的领域实体进行保存
            _domainRepository.Save(aggregate, aggregate.Version);
        }
    }

// IDomainRepository的实现类
    public class DomainRepository<T> : IDomainRepository<T> where T
    {
             // 并没有直接对领域实体进行保存，而是先保存领域事件进EventStor
        // 然后EventBus把事件分配给对应的事件处理器进行处理，由事件处理器来把
        public void Save(AggregateRoot aggregate, int expectedVersi
        {
            if (aggregate.GetUncommittedChanges().Any())
            {
                _storage.Save(aggregate);
            }
        }
    }

// Event Store的实现，这里保存在内存中，通常是保存到具体的数据库中，如SQL S
    public class InMemoryEventStorage : IEventStorage
    {
         // 领域事件的保存
        public void Save(AggregateRoot aggregate)
        {
            // 获得对应领域实体未提交的事件
            var uncommittedChanges = aggregate.GetUncommittedChange
            var version = aggregate.Version;

            foreach (var @event in uncommittedChanges)
            {
                version++;
                // 没3个事件创建一次快照
                if (version > 2)
                {
                    if (version % 3 == 0)
                    {
                        var originator = (ISnapshotOrignator)aggreg
                        var snapshot = originator.CreateSnapshot();
                        snapshot.Version = version;
                        SaveSnapshot(snapshot);
                    }
                }

                @event.Version = version;
```

```
                // 保存事件到EventStore中
                _events.Add(@event);
            }

            // 保存事件完成之后，再将该事件发布到EventBus 做进一步处理
            foreach (var @event in uncommittedChanges)
            {
                var desEvent = TypeConverter.ChangeTo(@event, @ever
                _eventBus.Publish(desEvent);
            }
        }
    }

  // EventBus的实现
    public class EventBus : IEventBus
    {
        private readonly IEventHandlerFactory _eventHandlerFactory;

        public EventBus(IEventHandlerFactory eventHandlerFactory)
        {
            _eventHandlerFactory = eventHandlerFactory;
        }

        public void Publish<T>(T @event) where T : DomainEvent
        {
            // 获得对应的EventHandle来处理事件
            var handlers = _eventHandlerFactory.GetHandlers<T>();
            foreach (var eventHandler in handlers)
            {
                // 对事件进行处理
                eventHandler.Handle(@event);
            }
        }
    }

 // DiaryItemCreatedEvent的事件处理类
    public class DiaryIteamCreatedEventHandler : IEventHandler<Diar
    {
        private readonly IStorage _storage;

        public DiaryIteamCreatedEventHandler(IStorage storage)
        {
            _storage = storage;
        }

        public void Handle(DiaryItemCreatedEvent @event)
        {
            var item = new DiaryItemDto()
            {
                Id = @event.SourceId,
                Description = @event.Description,
                From = @event.From,
                Title = @event.Title,
```

```
                    To = @event.To,
                    Version = @event.Version
                };

                // 将领域对象持久化到QueryDatabase中
                _storage.Add(item);
            }
        }
```

上面代码主要演示了Command部分的实现，从代码可以看出，首先我们需要通过ServiceLocator类来对依赖注入对象进行注入，然后UI层通过CommandBus把对应的命令发布到CommandBus中进行处理，命令总线再查找对应的CommandHandler来对命令进行处理，接着CommandHandler调用仓储类来保存领域对象对应的事件，保存事件成功后再将事件发布到事件总线中进行处理，然后由对应的事件处理程序将领域对象保存到QueryDatabase中。这样就完成了命令部分的操作，从中可以发现，命令部分的实现和CQRS系统中的系统结构图的处理过程是一样的。然而创建日志命令并没有涉及事件溯源操作，因为创建命令并需要重建领域对象，此时的领域对象是通过创建日志命令来获得的，但在修改和删除命令中涉及了事件溯源，因为此时需要根据命令对象的ID来重建领域对象。具体的实现可以参考源码。

下面让我们再看看查询部分的实现。

查询部分的实现代码：

```
public class HomeController : Controller
    {
        // 查询部分
        public ActionResult Index()
        {
            // 直接获得QueryDatabase对象来查询所有日志
            var model = ServiceLocator.QueryStorage.GetItems();
            return View(model);
        }
    }

public class InMemoryStorage : IStorage
    {
        private static readonly List<DiaryItemDto> Items = new List

        public DiaryItemDto GetById(Guid id)
        {
            return Items.FirstOrDefault(a => a.Id == id);
        }

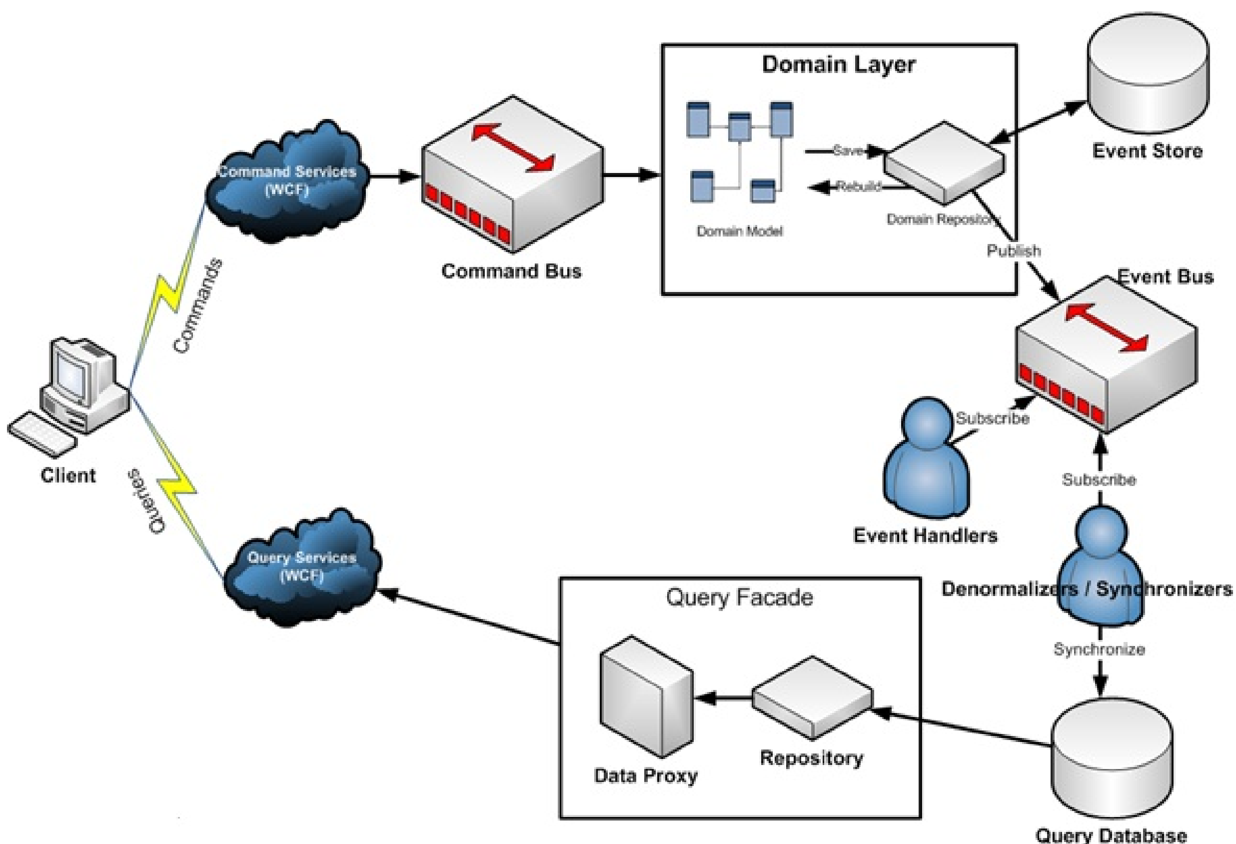
        public void Add(DiaryItemDto item)
        {
            Items.Add(item);
        }

        public void Delete(Guid id)
        {
            Items.RemoveAll(i => i.Id == id);
        }

        public List<DiaryItemDto> GetItems()
        {
            return Items;
        }
    }
```

从上面代码可以看出，查询部分的代码实现相对比较简单，UI层直接通过QueryDatabase来查询领域对象，然后由UI层进行渲染出来显示。

到此，一个简单的CQRS系统就完成了，然而在项目中，UI层并不会直接CommandBus和QueryDatabase进行引用，而是通过对应的CommandService和QueryService来进行协调，具体的系统结构如下图所示（只是在CommandBus和Query Database前加入了一个SOA的服务层来进行协调，这样有利于系统扩展，可以通过SOA服务来进行请求路由，将不同请求路由不同的系统中，这样会可以实现多个系统进行一个整合）：



关于该CQRS系统的演示效果，大家可以自行去Github或MSDN中进行下载，具体的下载地址将会本专题最后给出。

七、总结

到这里，本专题关于CQRS的介绍就结束了，并且本专题也是领域驱动设计系列的最后一篇了。本系列专题的内容主要是参考daxnet的ByteartRetail案例，由于daxnet在写这个案例的时候并没有一步一步介绍其创建过程，对于一些领域驱动的初学者来说，直接去学习这个案例未免会有点困难，导致学习兴趣降低，从而放弃领域驱动的学习。为了解决这些问题，所以，本人对ByteartRetail案例进行剖析，并参考该案例一步步实现自己的领域驱动案例OnlineStore。希望本系列可以帮助大家打开领域驱动的大门。

由于现在NO-SQL在互联网行业的应用已经非常流行，以至于面试的时候经常会被问到你用过的非关系数据库有哪些？所以本人也不想Out，所以在最近2个月的时候学习了一些No-SQL的内容，所以，接下来，我将会开启一个NO-SQL系列，记录自己这段时间来学习NO-SQL的一些心得和体会。

本专题所有源码下载：

Github地址：https://github.com/lizhi5753186/CQRSDemo

MSDN地址：https://code.msdn.microsoft.com/CQRS-1f05ebe5

本文参考链接：

http://www.codeproject.com/Articles/555855/Introduction-to-CQRS

http://www.cnblogs.com/daxnet/archive/2010/08/02/1790299.html

[.NET领域驱动设计实战系列]专题十一：.NET 领域驱动设计实战系列总结

一、引用

其实在去年本人已经看过很多关于领域驱动设计的书籍了，包括Microsoft .NET企业级应用框架设计、领域驱动设计C# 2008实现、领域驱动设计：软件核心复杂性应对之道、实现领域驱动设计和Asp.net 设计模式等书，但是去年的学习仅仅限制于看书，当时看下来感觉，领域驱动设计并没有那么难，并且感觉有些领域驱动设计的内容并没有好的，反而觉得有点华而不实的感觉，所以去年也就放弃了领域驱动设计系列的分享了，但是到今年，在博客园看到还是有很多人写领域驱动的文章，以及介绍了领域驱动设计相关的好处，这时候我就想，领域驱动设计真有这么好吗？但是我并不觉得好了，这时候就想是不是我没有实战没有深刻的感受呢？因此我在今年3月份的时候又重拾领域驱动设计，打算分享一系列关于领域驱动设计实现的文章，所以也就有了这个系列。

二、本系列所有专题目录

在刚开始打算写的时候，本以为对领域驱动设计相关理论知识掌握的不错，但当真正打算写的时候，发现之前的知识储备差不多忘的差不多了，无奈下只有重新再拿起书本来温习一遍，不过这次温习很快，因为之前都已经看过一篇。这里分享出来就是想告诉大家，没有真正实践过的东西是很容易忘记的，这时更加坚定了我要写这一系列的文章了。这个初衷也是我一直坚持写这个系列的动力。现在这个系列也告一段落了，从中我确实体会了领域驱动设计的美妙之处，以及现在软件设计的发展和改变。下面是本系列中所有专题的一个目录，为了帮助更好地收藏和自己进行索引，关于实践下来的体会将在下一部分分享给大家。

[.NET领域驱动设计实战系列]专题八：DDD案例：网上书店分布式消息队列和分布式缓存的实现

[.NET领域驱动设计实战系列]专题九：DDD案例：网上书店AOP和站点地图的实现

[.NET领域驱动设计实战系列]专题十：DDD扩展内容：全面剖析CQRS模式实现

三、总结

通过对领域驱动设计的实践，本人对领域驱动设计的有点和缺点都有了一个清晰的认识。并不是所有软件都适合应用领域驱动来实现的，例如在一些公司还是用三层框架来进行软件的开发，这样并没有什么不好，针对一些业务逻辑简单和后期需求变更不大的软件，完全可以使用三层框架来进行开发，因为三层框架尽管各层之间的依赖关系比较大，不利于扩展。但其好处就是简单，快捷。对于一些小型项目用三层框架是极好的。但对于一些大型项目来说，三层框架可能就不怎么适合了，尤其是大型网站项目。这时候就可以考虑使用领域驱动设计，领域驱动设计推崇的富领域模型，即将相关实体的业务逻辑放在领域实体里面。领域驱动设计思想分层结构更细，使得各层之间的依赖降低，通过引入依赖注入框架拉进入达到低耦合，高内聚原则。并且通过仓储模式，可以使得针对其他数据库的存储也可以很方便的进行扩展。采用领域驱动设计也可以更多实施测试驱动开发，早在以前的项目，哪里会有单元测试这个东西啊。

通过这个系列最深刻的感受，除了对领域驱动设计有了更进一步的认识外，还有一点更深刻的感受就是做软件的一定要把自己学到的内容实践起来，并且通过博文或其他方式进行总结，这样才能更好的积累。尽管通过博文的方式不经常用一样会忘记，但是很多东西你总结了就是和没总结的不一样，总结了可以对知识有一个系统的梳理，这样可以让你深刻理解知识点，尽管忘记了，它也是被记录在大脑的某个角度，当重新遇到问题时，你完全可以通过自己写的博文重新找回来，并且找回来的认识并不会比之前的理解少，可能更加多，但是不总结的话，那种忘记可能就是真的忘记了，等于没看一样。所以，对于做软件来说，真需要多实践。所以，还是奉劝大家可以多总结，多实践，抛下浮躁的心态，想做好技术，需要的静下心来专研和实践。最近，刚接触的一个项目用到了一个一些非关系数据的内容。所以接下来，我将会新开一个非关系数据库的系列来进行总结自己这段时间里的经历。其中包括Mongodb、Redis等非关系数据库的相关内容。

最后附上，所有专题的完整DDD实践案例下载地址：

DDD实践案例下载地址：DDD实践案例：网上书店